The Report Committee for Robert Gerard Nadon
Certifies that this is the approved version of the following report:

The Traceable Lifecycle Model

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor: _____
Suzanne Barber

_____
Thomas Graser

The Traceable Lifecycle Model

by

Robert Gerard Nadon, B.S.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2010

Acknowledgements

.

This paper was inspired by Dr. Suzanne Barber, Dr. Thomas Graser, and the University of Texas, ECE department. Further dedication goes out to those that have contributed to the study of software lifecycle models as well as the study of traceability. They contributed a large amount to not only this paper, but to the future of this field and to the future of software engineering in general.

November 22, 2010

Abstract

The Traceable Lifecycle Model

Robert Gerard Nadon MSE

The University of Texas at Austin, 2010

Supervisor: Suzanne Barber

Software systems today face many challenges that were not even imagined decades prior. Challenges including the need to evolve at a very high rate, lifecycle phase drift or erosion, inability to prevent the butterfly effect where the slightest change causes unimaginable side effects throughout the system, lack of discipline to define metrics and use measurement to drive operations, and no "silver bullet" [1] or single solution to solve all the problems of every domain, just to name a few. This is not to say that the issues stated above are the only problems. In fact, it would be impossible to list all possible problems—software itself is infinitely flexible bounded only by the human imagination. These are just a portion of the primary challenges today's software engineer faces.

There have been attempts throughout the history of software to resolve each one of these challenges. There have been those who tried to tackle them individually,

simultaneously, as well as various combinations of them at one time. One such method was to define and encapsulate the various phases within software, which has come to be called a software lifecycle or lifecycle model.

Another area of recent research has lead to the hypothesis that many of these challenges can be resolved or at least facilitated through proper traceability methods. Virtually none of today's software components are completely derived from scratch. Rather, code reuse and software evolution become a large portion of the software engineer's duties. As Vance Hilderman at HighRely puts it, "Research has shown that proper traceability is vital. For high quality and safety-critical engineering development efforts however, traceability is a cornerstone not just for *achieving success*, but to *proving* it as well." [2] So if software is not derived from scratch, having the traceability to know about its origination is invaluable.

Given today's struggles, what is in store for the future software engineer? This paper is an attempt to quantify and answer (or at least project a possibility) that involves a new mindset and a new lifecycle model or structure change that may assist in tackling some of the above referenced issues.

Table of Contents

List of Tables

List of Figures

List of Illustrations

Chapter 1: Introduction

Software lifecycle models are key aspects of virtually all software engineering enterprises within the industry today, though some organizations document and abide by the models more rigorously than others. These lifecycle models have wide variances, varying structures, and unique definitions, in a manner that is reflective of the corporations themselves. A lifecycle model pared down to its most rudimentary level is simply an encapsulation of various activities into a stage. With this encapsulated group, a transition from one group to the next allows for the creation of a model, and this model should represent the software cycling throughout its entire life, hence the term "lifecycle model." This lifecycle needs to be documented to keep things on course, and there are organizations, such as the SEI, that will come in and evaluate how closely you follow your documented lifecycle model.

In practice, these models were developed to define the various stages a software product goes through, because once defined it is very advantageous to isolate encapsulated activities to a particular phase. [3] If, for example, all testing were to occur at a set section of the project, say after development ends and before release activity begins, then scheduling is much easier. Having the sum of the time for all activities to be accomplished before testing (requirements, design, code, etc.) and a start time to begin the stopwatch, it would be mathematically possible to state when the given testing resources would be needed. Additionally, scheduling optimizations could occur as independent agents with the ability to be overlapped. This was how the lifecycle models came to be. Adding this model-based structure to any engineering project can give it an inherent ability to be improved. [4] The problem comes in that it is very difficult to place a generic model on the actual activities that occur for various software projects as every

project is unique and takes on its own life. They do, however, still contain many similarities and therefore can and should be encapsulated and structured.

Another key area of focus within the industry is traceability. Traceability gives an explanation of an artifact's origin as well as a description of the artifact. This is sometimes called a "footprint", in a reference to tracking a human or animal, such as the infamous Sasquatch hunters. The ideal footprint of an artifact has been a goal of the software industry for some time, and has been attracting even more attention lately. One of the original footprints was found within source code comments which were inserted throughout the software system. Other artifacts, such as documentation and architecture, may contain a reference section within them so that some basic information can be given to the artifact for traceability and credibility concerns. This variance though is also one of the problems within an all encompassing traceability mechanism. Every stage (even identical stages at different enterprises) has different representative forms of traceability.

This paper will attempt to resolve this issue of traceability representation variance, traceability avoidance, and undefined traceability content. By intertwining these two key aspects of software engineering together, they coincide and hence one empowers the other. The new model, the Traceable Lifecycle Model, would be a new model that could be used as a basis for software products ensuring the correct amount and form of a traceable entity be added to each artifact at each stage. This would have potentially numerous beneficial side-effects including greatly benefiting the seemingly impervious world of software evolution. Yet, at the same time, it should not overload the organization in overhead to the point of implosion.

Chapter 2: The Industry Gap

Software has been an industry since the middle of the 20th Century [5] and as of 2008 was an over $300 billion dollar industry [6], and the size has constantly increased. Add to this the fact that most software is not built from scratch, and the need for software to be comprehensible becomes apparent. There are numerous publications on the difficulty on maintaining software systems, and virtually all of them will explain how as software systems age their complexity increases and quality decreases. Lehman defines eight laws that govern the evolution of software as it is matures [7]:

1. **Continuing Change** – Software must constantly change to adapt with ever changing technology lest it become more and more unsatisfactory
2. **Increasing Complexity** – As software evolves it becomes increasing complex
3. **Self Regulation** – Software evolution processes are self-regulating with normal distribution of measures of process and product attributes
4. **Conservation of Organizational Stability** – Throughout the product lifecycle, global activity tends to remain constant
5. **Conservation of Familiarity** – Incremental and long-term growth of a software system tends to decline
6. **Continuing Growth** – Functional capability must be constantly increasing for customer satisfaction
7. **Declining Quality** – The quality of a system decreases as it evolves
8. **Feedback System** – Software Evolution processes tend to be multi-level, multi-loop, and multi-agent feedback systems

The above list of problems exponentially intensifies without traceability, which in turn greatly increases the risk that software system changes are not propagated

correctly.[8] The ideal solution to deal with the eight previous listed problems, would be to know what the original creators were actually thinking, when they developed whatever entity is now being altered. To dictate these thoughts in writing in a standard manner is the way in which the thoughts will be captured.

To see why traceability throughout the entire lifecycle model is a problem in today's industry, all one has to do is to take your favorite web browser and favorite search engine and query for "software traceability lifecycle" and attempt to muddle through the over 2 million results.

In addition to this mass conglomeration of information or "hits", there are many direct references which enlighten the reader on the size of the problem. Following are some examples that should help emphasize the gap.

Kirk Knoernschild explains that traceability of requirements is useful in both directions, helping the phases that came before and those that follow. He states:

> By linking the requirement back to the stakeholder, we are able to prioritize the requirement and determine how valuable a requirement is to a specific customer. Tracing the requirement forward allows us to understand and assess the impact of change, identify relevant artifacts that realize a requirement, and even determine if a feature of the system is needed depending on how much it is actually used [9]

Knoernschild is correct and one could further stipulate that this multidirectional benefit is true for any stage of the lifecycle, not just requirements.

Another area which frequently lacks traceability is reverse engineering. Reverse engineering and software evolution are becoming more popular and necessary than in the past. Though design and architecture can be extracted through a reverse engineering mechanism to obtain the box and line design, the aspects of traceability are lost during this reverse engineering process. There have been attempts, though unsuccessful, to

reverse abstract the traceability factors of an artifact [10]. This is a very challenging task because abstracting what a system does and why it does it are two very different problems, with the latter being much more difficult.

Requirements traceability alone can help save scope creep by having every action traceable directly back to the requirement from which it originated. This also helps the effects of post-phase requirements addition or modification. Another benefit of testing allotment is that the test cases may be directly tied to the original requirements so a traceable requirement is a more testable one. Even several areas of the United States government, including the Department of Defense and the Food and Drug Administration make requirements traceability a mandate. As stated by the regulatory commission: "A software life cycle model should be understandable, thoroughly documented, results oriented, auditable, and **traceable**." [11]

Traceability throughout the entire lifecycle model will not only enable decisions to be tracked and reversible, but will also have the benefit of forcing additional scrutiny of a decision so that unscrupulous decisions do not incorporate themselves into any portion of the system. Another facet that traceability throughout the lifecycle enables is reuse. An entity that has traceable decisions can be reused in a much more reliable manner than without. ASI's Datamyte's white paper on traceability and lifecycle management states how traceability goes hand-in-hand with lifecycle management quality. [12] Add to this the fact that traceability is not just a way of doing things right, but a way of proving it as well. With traceability, if something comes into question, the answer can be easily determined through ownership and verbal explanation found within the traceable footprint.

Another example can be found in the paper by Héctor García, Eugenio Santos, and Bruno Windels. That paper asserts that the latest technology within the world of

traceability discards many of the elements that make up a software product [13]. Each and every artifact should be included in the traceability. Most traceability discussions focus on requirements traceability and the ability to trace each artifact back to the requirement from which it was derived. However, there is much more to traceability, and those additional capabilities need to be tracked throughout the entire process and through each stage the software system goes through.

There is also a lack of tool support to fully embrace all the advantages of traceability. A white paper by Andrew Kannenberg from Garmin International and Dr. Hossein Saiedian from the University of Kansas, discusses why software requirements traceability remains a challenge [14]. That paper states that industry today is struggling to bring about traceability within the software lifecycle. They postulate that this is due to cost, understanding, and the ability to manage change.

Indeed, extensive research has been conducted in area of traceability. Why then do we not have more of an attempt to include such an important item in all aspects of software engineering? There are many reasons including time, costs, and schedule impact in addition to the fact that it is very difficult to have precise mapping of traceability across multiple stages of the lifecycle as each stage has its own unique set of artifacts, deliverables, and representations. This is further complicated by the fact that recording design decisions is cumbersome and takes time and resources and can also be instrumental in holding people accountable for the decisions they make. Many people shy away from recording their decision reasons because those reasons may invite a level of accountability that might lead to adverse career effects. Finally, management sometimes cannot see beyond the products immediate needs so traceability can be easily overlooked as a trade-off for things, such as time-to-market and budget throughout the various product stages.

Despite these fallacies, there have been attempts – with varying degrees of success -- to resolve these traceability problems. For example, the Bell Lab's Hyper Code tool has the ability to reduce code inspections, and this would be considered a very successful traceability mechanism. Also, there are several other tools, such as IBM's Rational which offers change and release management solutions, as well as, ValiMation which provides a database lifecycle management based traceability solution, as well as many others.

The make-believe story blogged by Eric Sink [15] summarizes a possible outcome from lack of traceability. This could happen at any organization, and could also be easily made into a Dilbert cartoon.

"A problem is reported to Joe in support, that in moving from version 6.0 to 7.0 there are several customers now complaining about a major delay. Joe goes to his supervisor, Sally, and together, after root causing, find out that this was due to a SQL table schema modification to simply allow for a person to be associated with multiple companies instead of just one. Unfortunately, several other areas of the code depended on a person only being associated with one company. So Joe and Sally wander off to find the responsible developer. They find it is Carl, and he does not really remember exactly why he made the change (that was almost nine months ago after all), but says he believes he was instructed to do so by QA. QA then directs them to a marketing person who after a short recollection states that it was necessary for a major sale and that it just was not implemented correctly."

So where does the problem lie? Should marketing not have made the big sale? Is there any point in this whole scenario where blame can be assigned? Now, where many organizations would look for a SPF (Single Point of Failure) to be the cause of the problem, here it is a lack of something instead of one particular person or area not doing

their job correctly. That something is traceability. To know the impact of a coding change before the change is made is an invaluable piece of information. This scenario could have been completely avoided if the ramifications of the change were investigated through traceable mechanisms. Even if no investigation or traceable reference was done, absolute minimal, easily automated traceability, such as ownership, would have enabled Joe and Sally to determine root cause in a much more expedient manner.

The fact is that there is an industry gap. The numerous industry cycles that have been placed on traceability throughout the entire lifecycle model is mind-boggling. This is not only a gap, but one that is in focus at this time within the software industry.

If there is such a gap, why haven't all software entities simply tied extensive traceability throughout all documents of all lifecycle phases? The answer is best described by the complexity of a software system. The problem lies in that a software entity has many complexities nicely summarized by Alexander Egyed; [16]

- **Exponential growth** in that there can be up to n^2 trace dependencies for n artifacts [Antoniol et al., 2002] [Card, 1992]
- **Non-linear increase in the number of software artifacts** during the course of the software lifecycle [Cross, 1991]
- **Syntactic and semantic differences** make it hard to identify exact traceability [Overgaard, 1998] [Jacobson, 1987]
- **Many-to-many mapping** of the phases. For instance, a requirement is often implemented by many design modules and design modules can implement many requirements. [Tilbury, 1989]
- **Incompleteness and inconsistency** exist throughout the different stages [Lindvall and Sandhal, 1996]

- **Different stakeholders in charge** of different artifacts where no single stakeholder understands all of them and each has their own definition of the traceability required [Boehm et al, 1998]
- **Increasingly rapid pace of change** and traceable paths throughout the system can change as the system evolves [Moore, 1995]
- **Imprecision and uncertainty** are typical description attributes of software artifacts [Finkelstein et al, 1992] [Egyed, 2004]

Even the industry's top guides to correct software creation, such as IEEE Standard 830- 1998, CMMI, and ISO 9001 mandate traceability as the right thing to do. [17] [18] These problems emphasize the need for traceability to be a fundamental part of the software engineering paradigm.

Chapter 3: The Life Cycle Model

Most all enterprises within today's software industry have some sort of lifecycle model that they abide by, and many have this lifecycle documented. There is a large variety of model forms that these companies attempt to follow, each of which seems to focus upon a particular area or goal. However, rarely are they followed exactly. Instead, they tend to be more of a guideline. When they are followed rigorously they run into problems due to the rigidness of the model and the uniqueness of a software entity. Unfortunately, since they are frequently just guidelines, *lifecycle drift* can be of major consequence producing butterfly-type effects, where the smallest change will be enormous consequences. Lifecycle drift is a slow change from the originally specified lifecycle, many times to adapt to an ever evolving and changing environment. That would lead one to believe there is a problem in the level of structure or flow within the model. However, this would be merely masking the problem, because if it is a requirement that there is some drift away from the model to accommodate the model, then the model should be able to account for such behavior. To fully explain this hypothesis a brief history of lifecycle models to point out some or their advantages and disadvantages is informative.

Most people attribute the original model to be the Waterfall Model. From there they branched to other variations usually directing attention to a particular specific need that the Waterfall Model or previous model failed to accommodate. There are numerous models that are in existence within the industry today, and this paper will attempt to give a brief description of some basic ones, starting with the original model.

THE WATERFALL MODEL

The Waterfall Model is so named because it takes on the appearance of a waterfall. Each step of the Waterfall Model must be completed before moving on to the next step. The order of these steps or phases generally starts with requirements. From there, once the requirements have been completed and all requirement artifacts have been approved, the architecture is created. Logically following the architecture, the next phase is the design. Following the design is the actual implementation, often coined as "coding" in the software industry. After the product has been built and the coders have compiled and completed their rudimentary testing, the validation phase is entered. This phase is also known as test or system test, and is the phase whereby the system undergoes any test to verify that the system is what was desired. Upon completion of all testing, and when an acceptably few number of bugs exist, the product is released. After this stage, a maintenance phase is entered. During this phase, customer use provides feedback to the product and the necessary maintenance tasks are performed. The Waterfall Model can best be illustrated through a picture like the one seen in **Figure 1. The Waterfall Model**.

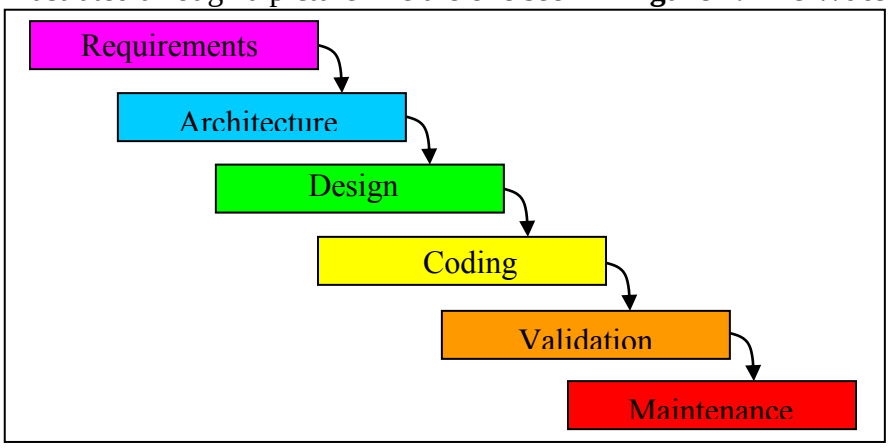

Illustration 1: The Waterfall Model.

Within the Waterfall Model each phase should be complete before going on to the next phase. Doing otherwise would not be practical because if any stage exhibits a problem or bug with the previous stage, the problem phase must revert back a step which is not accounted for in the model. This was one of the big drawbacks to this original model, and it quickly led to the next model, which is the modified Waterfall Model.

THE MODIFIED WATERFALL MODEL

The Modified Waterfall Model is the same as the Waterfall Model except that it accounts for the need to traverse backwards as well as forwards. So like the original Waterfall Model above, as described in Figure 1, it would be the same picture except that arrows also go back to the previous step. This accounts for the future steps finding problems with the previous, such as testing finding a coding bug. This function points out the inherent need for reverse tracking.

This model is much more realistic and is truly idealistic in that from one phase you can only move to a subsequent or preceding phase. Ideally after getting out of an early stage, like requirements, one would not want to have to revisit them from a later stage, such as validation. This unfortunately is not always how the phases go in the real world. It is common to find new previously unknown requirements and architectural problems during validation. Though undesirable, this is the way things sometimes work.

To avoid the rigidness of the Waterfall Model, a new model, called the "Spiral Model" was created.

THE SPIRAL MODEL

The Spiral Model was developed for risk-oriented projects; it works best for those projects that have poorly specified requirements. The Spiral Model, just as it sounds, is a

spiral that goes through quadrants starting with the most generic specifications and each revolution, spiraling toward a more specific set of concerns or risks, is addressed.

As is the case in the Waterfall Model, the Spiral Model can best be seen in an illustration. For this please see **Illustration 2: The Spiral Model**.

It is sometimes the case that a customer would want an early model or prototype of the system to ensure that the original specified requirements were accurate and reflect what the customer really wants. The Spiral Model can provide this, but the Spiral Model is more directed at tackling risks. If a prototype is desired very early on and continuously throughout the lifecycle of the product, then this maps well for the Rapid Prototyping Model.

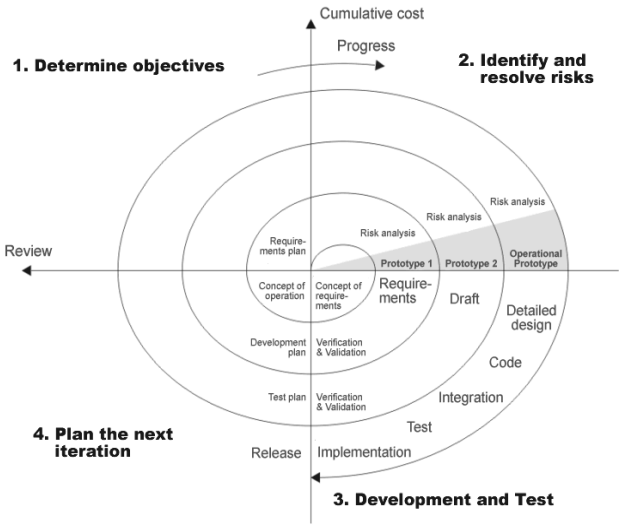

Illustration 2: The Spiral Model. [19]

RAPID PROTOTYPING

Rapid prototyping essentially creates several systems with each one inching closer to the desired end product. This also is addressed in Fredrick Brook's "plan to throw one away" mentality. The concept is that users are allowed to evaluate the developer's

concepts with a prototype. The users then tell the programmers what needs to change and the developers create another prototype.

THE V-SHAPED MODEL

The V-Shaped Model was developed because there was a desire to keep the Waterfall Model, but there was a need to test or validate each phase. So instead of having a single validation phase after coding, validation was done at every step, leading to the V-Shape Model illustrated in **Figure 3. The V-Shaped Model**. This model has a focus on validation including important steps frequently omitted – those steps usually used in designing the test suites. Test suite design is usually just categorized with the validation or testing phase but is truly a different activity than performing the actual tests.

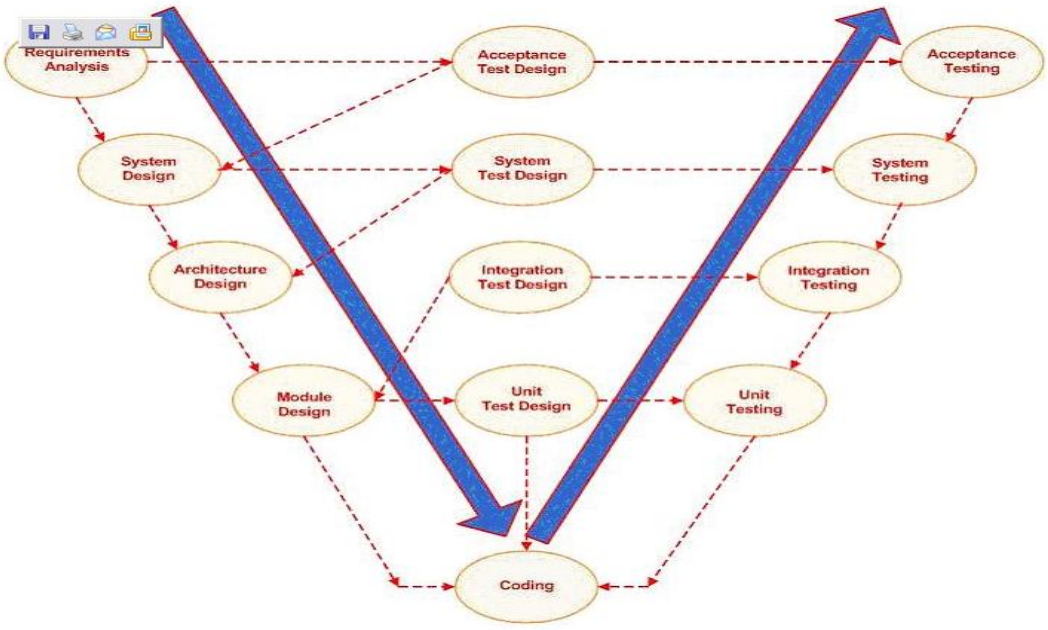

Illustration 3: The V- Shaped Model. [20]

AGILE

The latest craze is a lifecycle model that barely forms that which can be graphically depicted, Agile. Agile is a lifecycle model that is named after an adjective defined by the ability to move or think quickly. The Agile lifecycle model then prides itself on the ability to change to changing requirements without hesitation. The Agile lifecycle itself is more of a description or attribute of another lifecycle model, such as eXetreme Programming (XP), Scrum, Adaptive Software Development, Feature Driven Development etc. The Agile method itself is based on rapid and continuous delivery of software. It can be described best by a set of base principles summarized from the class lecture notes of the UT software measurement expert, Herb Krasner [21]:

1. Satisfy the customer through frequent software deliveries
2. Welcome changing requirements
3. Customers and developers work side-by-side
4. Build projects around motivated people
5. Face-to-face is the best form of communication
6. Working software is the primary measure of progress
7. Don’t do something that doesn’t add value
8. Teams should be self-organizing
9. At regular intervals teams should re-evaluate their effectiveness

OTHER POPULAR LIFE CYCLE MODELS

There are many more models within the industry including:

- **The Incremental Model**
- **Evolutionary Prototyping Model**
- **OSS Development Model**

- **IID**
- **RAD and RUP**
- **Legacy Maintenance**
- **… and many more**

The above list shows some of the models, and there are many more, each with its own strengths and specialties.

THE CHANGE

The amount of effort expended on lifecycle models in an attempt to accurately capture and/or direct the true life of a software product can be clearly seen within today's software industry. However, stepping back and adding infrastructure that allows verification of compliance to the model as well as the ability to traverse backwards would be extremely useful, possibly to even redo a portion of the lifecycle in case a post-stage problem is found. This and more can be achieved through the addition of correct traceability throughout the entire lifecycle to any model.

As one can easily see there is quite a variety of lifecycle models, and the models themselves are continuing to evolve and produce new variants based on new challenges. This evolution is due to the correct form of lifecycle drift described at the beginning of this section. For instance, it was noticed that requirements were continually changing and this was becoming a major deterrent to the entire effort. So Agile was discovered to not only accommodate but actually welcome this need for ever changing requirements. The industry should stop or even slow down this evolution. Instead the evolution should take a more engineering-based discipline and become smoother and more calculated. To do this we must have a completely documented and measured model with rationale for each aspect of it. This should be documented in the traceable footprint, and this would

then allow for the given model to evolve with predictable and justifiable reason and direction.

Chapter 4: Traceability

So what is "traceability"? In the most rudimentary definition traceability is the ability to be attributable, or capable of being traced. For example a comment found within source code was primarily added to compilers as a traceability mechanism to allow for code reasoning and ownership. There are also side notations in design and architecture components, origination notion in requirements, and various fields in bug-tracking tools. This intelligence notation is a side note from the main functionality of the component used to serve as the traceability of it.

So what is good traceability? A survey was done within the industry by StackOverflow entitled: "What was the best comment you have ever encountered survey?"[1] The following are some of the answers:

1. return 1; #returns 1
2. I did this the other way
3. When I wrote this, only God and I understood what I was doing. Now, only God knows…
4. Sometime (SIC) I believe the compiler ignores all my comments.
5. For the brave souls who get this far: You are the chosen ones, the valiant knights of programming who toil away, without rest, fixing our most awful code. To you, true saviors, kings of men, I say this: never gonna give you up, never gonna let you down, never gonna run around and desert you. Never gonna make you cry never gonna say goodbye. Never gonna tell a lie and hurt you.

Though most "up until the wee-hours" programmers can find humor in the above

[1]The Complete results of this survey can be found at http://stackoverflow.com/questions/184618/what-is-the-best-comment-in-source-code-you-have-ever-encountered website.

referenced comments, outside of that there is very little value to help in the comprehension of the code itself. An interesting statement about our industry is that when asked for the "best" comment ever encountered all examples were the epitome of worst. They give neither insight into the code or the developer's thoughts while constructing it, nor any useful tips to anyone wishing to modify it. So what is a good comment? A good traceable comment should give the future reader understanding and value. The comment can be measured by the depth of understanding and usefulness it brings to anyone looking at the artifact. The practice of software evolution teaches us that a "good" traceable comment is one that allows for those individuals, who alter the system in the future, to be able to have the deepest level of understanding of it as possible.

Though comments are a large portion of software traceability there are many more artifacts than that to a substantial piece of software, and so there are many more aspects to traceability. The software requirements section was the first area of software to coin the term "traceability" and the need for a footprint to give value to the corresponding phases. Each stage has tended to focus more on a specific area of knowledge to attribute the footprint:

- ⇨ Requirements – Ownership (the who)
- ⇨ Architecture/Design – Rationale (the why)
- ⇨ Coding – Functional Structure (the what)

More and more, today's engineer realizes the value of having all aspects at all levels. Having a rationale as to why a requirement is set gives it more value and comprehension, as well as with rationale, there tends to be more acceptance, or at least understanding, for the future stages as to the reasons for the original requirement's request. Traceability not only includes most aspects at every level, but further that they

be consistently represented across levels. This can be accomplished through either a central traceability expert guiding the traceability or adequate standards and guidelines for this traceability, either of which is inexpensive relative to the great benefit that is obtained. So what is the makeup of a good traceable footprint?

TRACEABILITY SUBSTANCE

The make-up of a good piece of traceability can be comprised of a variety of different attributes in which this paper will create taxonomy for understandability reasons. In essence, for software engineering, we wish for every stage or phase, previously described in the lifecycle section, to be traceable. The footprint for every artifact and every decision should define more than just a person's name or functional structure. The footprint could describe several aspects of the document, such as the following eight categories.

- ✓ ID – This is a unique identification stamp given to the component. This stamp could be automated, and can either be a simple unique stamp or have meaning, such as Arch01 for the first architectural element.
- ✓ Owner – This will be a time and ownership stamp. This can also be automated, such as with user ID and system check-in time.
- ✓ Rationale – The essence of the creator's thoughts.
- ✓ Sign-off – Verification of the traceability links from the previous stages.
- ✓ Description – This would show a functional trace of how each artifact was designed, which is very useful in recreating and reusing certain aspects of the system.

- ✓ Origin – This will define the component from which this component originated. Though the links already show this, this information can give even more of the impact of modification to the stage.
- ✓ Cohesion/Coupling – The evolution of cohesion and coupling can give insight into why certain artifacts seemed to have such undesirable levels of cohesion and coupling.
- ✓ Other – Industry proprietary field set up within the footprint. With this evolution or origination it can be clearer how anything came to be, even that which is meaningful only within the organization itself.

The conglomeration of this information will be defined as the traceable footprint. Should an engineer add everything each and every time that any piece of software is created? The answer is it simply: it depends. The choice of which of the above possible attributes will be added should be customized to the product and those who will be tasked with its evolution. For ongoing projects and products, Return On Investment (ROI) should be measured to obtain optimal footprint reference only including that which the time spent creates value greater than that of the time to create the footprint. For the sample found in Appendix A, the following table can be used as a guideline.

Attribute	Use
The ID	Identification of the footprint
The Who?	Ownership reference
The When?	A time stamp and version reference
The What?	Artifact's purpose and functional structure
The Why?	For reasoning, also sometimes called rationale
The With?	Functional interaction with other components
The Where?	Origin of this component

Table 1: Traceable Attributes

These attributes about an artifact and/or decision all have importance in the evolution, accountability, and credibility and usually should all be included, especially for large (100,000+ LOC) projects. Though the sample project is clearly not in that category this paper still uses a large portion at each stage for illustrative purposes. The above table includes seven essential groups that could and usually should be a part of the traceable footprint within the artifact.

Next will be a description, in further detail, of the above seven categories by taking them in groups. ***The ID*** of the element only has the requirement that it must be unique and somehow have access or directions to get to the footprint. This element may or may not give additional insight into the artifact that it represents. For example, a unique identification-stamp, such as an incremental number with a prefix of the stage is a frequent ***ID*** field. So for the 82nd requirement, ***The ID*** may be Req-082. Its use is comparable to that of a primary key in a relational database table in that it is used for

identification and organization. For automation of many of the advantages of having a completely traceable lifecycle, this section becomes a requirement.

This ***ID*** along with ***The Who*** and ***The When*** are grouped because they can easily be automatically inserted into the footprint through things, such as templates and tools that can use user ID and system time.

The Who is going to place a name or organization for both contact purposes if and when the need arises to question or query the artifact. This also gives ownership attribute to the particular component, similar to an author of a whitepaper or book. Also, if desired, a method of contact can be listed here as well as references to any snippets of IP, Intellectual Property, taken from other sources. This not only gives understanding of how the artifact came to be but also can be used for legal purposes as IP is becoming more and more of an organization value that is protected via legal means.

The When can be as simple as a time-stamp that could be, for example, placed upon check-in of the artifact to the central repository. However, in certain circumstances more elaborate time-stamping is needed, such as the times when the problem was found, root caused, and resolved. Regardless, ***The When*** question puts a time perspective into the component and can help show the evolution within the artifact. If, for instance, this piece of code is part of a system that has a major update every year, then the year of the time-stamp could give a release-based traceability as to when the component was added or altered.

The next two traceable attributes, ***The What*** and ***The Why,*** are many times what one would find today within the coding stage of a software product.

The What gives a detailed explanation of functionally and structurally what was created or changed. In the first instantiation, the artifact needs to include details on how an artifact behaves. This is also not an excuse to create incomprehensible artifacts. The

artifact itself should be inherently intuitive as how it does what it does, while this portion of the traceability should give additional details such as any particular technique or especially any shortcuts that were taken. Above that, an overall description is in order. For example, for a function all of the data elements used – both those that are instantiated within the artifact and those that are passed to it – the functions performed and responsibilities of the artifact, and the various paths (both data and execution) through the function could be described. This section can easily balloon into so much information that it will be completely ignored and create a negative ROI. To avoid that, one technique that can be employed is a system of reviews wherein another individual in the industry who is not on the project could take the artifact and see if, with the given amount of traceability, they can understand the functionality and structure of it.

For modification aspects of software engineering there are tools that can provide "diffs" of a before-and-after which can be a very useful portion of ***The What***. Though this "diff" gives a basic description of what was there before and after the change, this also leaves details out. Essentially, ***The What*** needs to also be a clear path to unwind any change that has been done. Explaining exactly what the change is supposed to accomplish through a spoken language will give further insight into what the particular component is and does.

The Why or rationale will be detailed next, and is also a key aspect of the footprint. To stress its importance, the rationale could be taught as a science all on its own. In the past there have been forced mechanisms that tried to create rationale within a particular phase. For instance, upon checking in source code, many times a rationale box comes up with a reason for change description, that does not allow for blank entries; however, incorrect use of rationale is the case where the traceability is recorded as statements such as, "I did, what I did" or "Doing this change because I was forced to."

That is not what is meant by traceability and it is a problem with trying to force someone into giving rationale. Instead of the poorly worded above examples, a statement such as, "Due to track bug 17-074, I modified the workAssociate SQL table to now account for a person to be tied to multiple companies as this was the original instantiation of the person-to-company relationship. This leads to a change in the design of the WorkForce component on the HireIn module. A search was then performed for other instances using the workAssociate SQL table and a message was sent to the owners of the particular modules to ensure no side-effects were encountered." A note like this or something along those lines provides valuable information and therefore becomes a valuable traceable facet.

A major problem with ***The Why*** or rationale is that in general people, especially engineers, do not like to give reasoning as to why they do something. It can be seen as lack of respect or appreciation to have to give the reasons why something was done. This leads to engineers frequently trying to be humorous in this section to avoid the discomfort of having to explain their actions. To fix this is where a change in mentality needs to come into play in the software industry. There can be found an air of egotism in engineering especially prevalent within the software engineering arena. This should be improved through a method of rationale standardization as well as an alteration in the general mentality of the software engineer, which will come with the industry's maturity.

That leads to the final pair and that is ***The With*** and ***The Where***. These last two are less commonly found within standard traceable artifacts; however, they may indeed be the most useful.

The With will be any type of interaction with other components. For instance, any calls to other artifacts within the same stage, such as calls to other external components that are not a part of the component currently being attributed with

traceability. Add to this, any input and output of the artifact to enable a clearer picture of where the artifact fits into the system. Essentially, with this section of the footprint, an easy comprehension of the artifact's cohesion and coupling should be ascertained. This field has the extra advantage of forcing the editor to "think outside of the box" in that it forces the user to describe all elements that interact with this artifact. Frequently, problems arise in that each and every piece is done correctly it is simply a matter of them failing when all used together. It is a very simple trap to fall into focusing solely upon the entity that is currently being worked ignoring those entities with which it interacts. This portion of the footprint helps limit that problem.

The Where is the backbone of the traceable lifecycle and should be included within each component, if many of the listed advantages of the traceable lifecycle are to be actualized. ***The Where*** is a field that lists the artifact(s) which originated the component. So, for example, if requirement X directly leads to implementing deign component Y, then Y will have a ***Where*** field pointing to X. This mapping is a many-to-many mapping, and can be from any artifact in which the given one was derived. This is frequently the previous stage, for example, in the waterfall model the code can be directly linked back to the low-level design specification which originated from an architectural module, which came from a given set of requirements, which were written to document the original customer specifications. There can be exceptions to this though, in which items can be derived from outside the standard lifecycle instead of directly coming from the previous stage. For example, certain architecture components are necessities of the system and do not have a requirement to guide them so no origination link would exist (though some people create this as a dependency requirement and then the link can be made back to it). With this ***Where*** component, a graph can be built and used to guide the impact of changes and show a mapping of relationships between all the other footprint

fields. It can illustrate change-impact, in that everything that the component is connected to should be visited to ensure that the change did not proliferate and affect it.

With these seven possible attributes we now have the make-up of the footprint. In essence, anything that would help describe useful aspects of the entity and create value with positive ROI should be included in the footprint. So even if there are other areas within an organization or endeavor that would have perceived value to document, but are not included with the previously listed categories, they should also be included. This footprint, much like other areas of software engineering, must be flexible in that each situation is going to dictate a slightly different content and appearance of the footprint. Next, let us go into detail on how to determine whether the various sections should be included.

TRACEABILITY MEASUREMENTS

So how much traceability do we incorporate into our project? The answer to this question is like many questions in this industry: It depends. Each and every software project seems to take on its own unique life and therefore has its own unique traceability needs. Preferably done through a designated traceability expert, measurements should be taken to find the value and effort required for the various factors of the traceability.

To define the complete footprint for every decision within the artifact is quite a cumbersome task, and further research needs to be conducted to find out exactly what is the best form of representation and linguistics to use for traceability. It should take into account the costs and value of creation, such as time, use, understandability, validity, ease on the eye, and practicality.

Please note that too much traceability can also be a problem. A project can get crushed under its own weight, as an old Shakespearean adage states, “Too much of a

good thing is a bad thing." So defining how much of the traceability to use needs to be customized to the situation. To determine this we need to use the traceability that provides value and has a positive ROI. It is a difficult task to project future ROI, as frequently, from the inception of a software development project, its future use is not yet known. If it was known, then only successful software products would ever have been developed, which we all know is clearly not the case. Given this challenge, there still are measurements which can help us guide the amount and which traceability attributes to include. Unfortunately, with traceability the measurements are generally past-looking as oppose to future-looking. Regardless of the direction, the measurements are a needed piece to allow an organization to accurately determine which traceability is valuable and which is costing more than it is worth.

So then how do you measure software traceability? One method, behind defining a metric in which to measure anything, is to decide on what the desired outcome is and place a numerical value on that. For example, we can track each of the main categories which comprised our taxonomy of traceability in Section 4.1 and also an overall measurement for how those components amalgamate.

The measurements need to be done both for overall and the individual categories. For an overall perspective it would be ideal to have a consistent metric used. For instance, if throughout the requirements phase a first and last name were used for identification and at the design phase a user ID was used, this can lead to inconsistency and difficulty tracing the entire project especially at a stage late in the game such as product retirement analysis, unless overall project traceability oversight has been conducted. Such oversight should be managed through a central location, such as a database, to incorporate all of the traceability for each phase, whereby a user ID can then be translated into a full name and contact information. Furthermore all predefine

traceability artifacts should be stored here and put in a consistent manner. This traceability database may seem like an overwhelming burden, but it can prove itself invaluable especially at the later stage of a product if major architectural modifications are in order. Furthermore, having a quick re-useable traceability database with the most basic table structure is not that much to undertake, especially if it can be automatically updated from the repository check-in engine. So for an overall measurement we are looking for consistency and centralization. So a number can be calculated, such as the number of consistent traceable footprint definitions divided by the total number of footprints. There are many more variations to this formula that could and maybe should be used, and that is due to the variant nature of software, and should be added by the individual enterprise. In order to find whether these additional efforts – measuring overall traceability and storing them within a database – actually produce a worthwhile positive ROI, one could use time spent within the database as the measurement. The time could be calculated by taking the time the database was used for looking up traceable information divided by the time it took to add/update the information. Mathematically, it would look like: $^{\text{Use-Time}} / _{\text{Creation-Time}}$. This calculation will give an ROI factor, and a greater than 1 result should be interpreted as positive ROI.

For ***The Who*** portion of the of the traceability we would like to have the ability at any stage including product retirement to be able to find out the original author of an artifact along with any of those who made any modifications therein. Now the first metric should be defined as the overview of such a statement. For each artifact, we should have an identification of anyone who contributed to it. This could be given as an average of the missing contributors, such as a metric of the Σ (missing contributors / artifact) divided by the total number of artifacts. However, we may run into problems as it is usually difficult to define a "missing" contributor. This can be accomplished through

use of automation, such as a repository for each and every artifact and upon check-in requiring that identification be given or using the users ID. If the organization does not have such repositories, a different metric should be defined. No matter what stage the software product is in, a collection of all artifacts could be sampled (either randomly without replacement or stratified) and tested to verify that the owner can be determined, found, and is contactable. The percentages of sampled components that have a traceable ownership, then becomes an ideal metric for ownership.

The next category we could employ to ensure that our traceability adequately reflects adroit artifact explanation is that of time-stamping or ***The When***. This of all areas should be easiest to consistently automate. A time stamp is simple and has many advantages. If it is known when the artifact was added or modified, we can then identify the times when each of the various changes or additions were added and therefore attribute those changes to a specific version. To measure this, an effort could, and maybe should be conducted at release time. That process would involve taking each phase and placing a categorization as to when (which version) the modification is attributable. Also recommended is avoiding an easy trap that uses the original release as a default if no other version has been attributed. This can be inaccurate, and an inaccurate measurement is more dangerous than no measurements at all. Also, certain organizations are doing modifications to different versions at the same time, in which case a separate traceable category of release is recommended. Again for any measurement an ideal use is to determine the exact ROI that is received and to optimize that. To accomplish this, since this is almost assuredly automated, would be to measure the effort to create and incorporate this automation verses the value of the attributed release identification described above as well as other knowledge that a time-stamp can give us, such as time-based inconsistencies. Generally, it is safe to assume that the simple endeavor of adding

a consistent time-stamp to the artifact is almost assured to be a positive ROI as so little effort is required to instantiate it.

The next category is ***The What***. Again, one should be thinking about the desired outcome of that which we wish to accomplish by adding this category to our traceability. ***The What*** stage is supposed to give us the description of the purpose of the given artifact along with the functionality and methodologies used to accomplish this purpose. To measure this we should get an understanding of a given component or artifact's purpose and the methodologies used to accomplish this purpose. That would lead to two metrics one for purpose and one for methodologies used. These metrics can be captured by using Likert scale fashion techniques where a general feeling is given as to the purpose and comprehension of methodologies of each component within that phase. This could also be used as a good sign-off checkpoint that could be created to gate exiting a phase, thereby helping to mandate proper traceability. If an outside engineer cannot understand the artifact, given both the artifact and the footprint, then the artifact would need to again be visited, as most likely there are other areas outside the footprint that are inadequate.

From here we encounter one of the most difficult categories to measure and keep consistent; that is ***The Why***. ***The Why*** for each phase can almost take on a unique style and meaning of its own. ***The Why*** for a requirement could be a strong demand from a customer, while the next requirement could have ***The Why*** section that explains the process by which it came to be, such as lessons learned from previous projects. Furthermore, ***The Why*** from a code segment or test suite may have a clear and insightful purpose as to the reason for its existence while the next segment has no explanation whatsoever. Here again the end goal for this task is to produce a worthwhile value or positive ROI as a result of the effort expended. While most of the traceability gains do occur after the effort is spent to create them, sometimes long after, this particular

category can have some immediate benefits in a similar manner as ***The With*** does. ***The Why*** can serve many purposes including that of ensuring proper thought was given to the particular addition or modification. Additionally, future benefits can be priceless as to understanding of the thoughts behind the reasons for the given addition or modification. Stated earlier in this paper was that our ideal situation would be to be able to see into the mind of the author during the artifact creation; this category is the closest category to accomplishing this feat. Furthermore, ***The Why*** can be absolutely required in adding functional capabilities would seem preposterous to the original author but were added there for a very specific reason. This is not always the case and this category frequently gets intertwined with ***The What*** category. The key difference here is that ***The What*** tells us the purpose of the artifact and how that purpose is accomplished, where as ***The Why*** is more of a defense for the artifact's existence. The measurements for this category could therein be measured by sampling. It could be directed at someone in the future being able to read ***The Why*** and be able to justify as to why the artifact was added or modified. This should not be measured with the expectation that the future reader will agree with the reasoning. Its purpose should be simply that the reader understands the reasoning. It may be the case that adding the reasoning even gives the future maintainer adequate reason for a modification or removal of the entity.

The next stage to be measured is ***The With***, which gives us the measurements for how well we captured our interaction with other entities. This can be defined as the systems coupling and cohesion. Coupling and cohesion are known attributes and have well defined metrics by which they are measured. Cohesion is a value of how strongly related the various objects are within a component or artifact. Coupling, on the other hand, is how dependant the components are on each other. In general, it is desirable to have high cohesion and low coupling. Cohesion can be measured by the relations

internal to the object divided by the total number of relations. So to get an overall value of cohesion for the entire system, a mean can be determined simply through direct formulation: $Ch = 1/n \ _{i=1}\Sigma^{n} \ Ch(C_i)$. This is computed by adding the values of the cohesion for each object and dividing by the total. For coupling, a very similar mechanism can be established that states that coupling is defined as the number of external relations for an object divided by the total. This is essentially an inverse to the cohesion metric. Similar mechanisms can be employed to obtain system-wide coupling.

The last phase will tie the different phases together; it is called ***The Where***. This is different than the previous category of ***The With*** in that this category defines things across phases instead of within them. In moving from one phase to another in any lifecycle there needs to be some sort of traceability in that the outcome of the previous phase somehow influences the current one. For those who challenge that this is not the case and that the phases can be completely independent, one should challenge the reasoning behind placing them into separate phases to begin with. If two consecutive phases are completely unrelated and independent, the definition of a lifecycle model stipulates to combine these steps into one. Since one phase in some way influences the next, this influence should be captured and recorded as this gives us a very nice recorded flow of product evolution, which in turn gives a much deeper understanding of the product. To measure the success of ***The Where*** section of traceability, we can count the number of objects or artifacts that do not have any traceability to a previous stage (l_p). We then take this number and add it to the number of artifacts that do not have a link to a future stage (l_f) and divide by the total number of artifacts. Formula-wise we get:

$$\textbf{\textit{The Where}} = (\Sigma \lambda_{\pi} + \Sigma \lambda_{\phi}) / \text{total}$$

When we create the connection diagram described in the later in this paper, this counting becomes easy and quite apparent.

Taking the lifecycle as a whole as well as dissecting it into categories, either as described above or a unique taxonomy based upon your own personal needs, gives us the specification to define a metrology strategy to guide in deciding exactly which traceability to include. Just as in virtually all successful business adventures, the decisions come down to keeping that which generates a positive ROI and discarding that which does not.

TRACEABILITY IMPLEMENTATION

There are various forms and methods to include the traceable information. These methods vary not only across different stages, companies, and enterprises, but also sometimes within the same group of an organization across different projects. Briefly listed are some of the various styles that personal research and experience have uncovered, and then suggest a conclusion to help guide the reader in their search for the right solution.

- First, a common method is to have a unique footprint incorporated throughout each artifact of the each component or stage within the lifecycle. So, for example, for the coding cycle, each file would contain comments both in the beginning and throughout the file. While within the design module the comments would again be at the top and strewn throughout the module in a callout fashion.
- Another method is an inclusion of various templates. These templates could include fields for the owner, a date, the origin and necessary sign-off list, the given functionality of the artifact, and a list of various decisions that were encountered, and the direction taken with reasoning for that direction. Essentially, this would cover the basis of the previously listed footprint's needs and help to ensure that there is consistency throughout the phases.

- Instead of using templates, the traceability could consist of having a complete collection of documents that includes all the traceable information in one central location. This becomes a central traceability library for all of the various cycles.
- A combination of these styles is frequently used and that the given phase determines the format of the footprint, more than the particular project does. This is also reinforced by the fact that many organizations break down their organizational structure in accordance with these stages. For example, the coding, completed all in one building, will be done through standard comment header and required minimal comment/line-of-code percentage, while maintenance, which is done in a completely separate location by a different part of the organization, uses a tool that generates a template based on a set of fields to answer and store the traceability, and a certain percentage of these fields are mandatory.

A variety of these mechanisms can be used for the traceable lifecycle model; however, it is necessary to give caution here as the more disjointed the various phases are, the more likely the interconnecting traceability will be lost. If the decision to use different mechanisms is chosen, then exact mappings should be defined and listed in order to tie the various footprint categories of the different stages together.

There are several components that have traditionally represented traceability in different forms. These components include: requirements, design, architecture, version control, maintenance issue tracking, marketing research and specifications, test suites and their results, and help-desk tickets, etc. An example of footprint representation of some of the previously mentioned waterfall-based stages can be seen as follows:

- Requirements – A template or table at the top of each artifact is frequently used. It contains the data that links the documents together with ownership, sources, and perspectives. Furthermore, callout notation is found with the graphical

requirements documentation such as operational models, entity relationship diagrams, operational reference models, and task hierarchies. Add to that in-code comments can also be found within formal and automated requirements such as those produce through the Use tool. [22]

- Architecture – Both prescriptive and descriptive style architectures could represent traceable artifacts. Templates could again be used to dictate the requirement(s) in which the architectural component or connector was derived. In addition, this template needs to contain the rationale behind each decision.
- Design – There are tools which help guide design traceability, such as gIBIS by Gaia [23], as well as other forms, such as notations and call-outs. Furthermore, check-in style document control traceability is used as well.
- Code – Source-code control mechanisms as well as comments that are used as a heading and additionally strewn throughout the code are frequently used forms of code traceability. These source-code control mechanisms can have check-in as well as check-out traceability components.
- Validation – Bug databases and tracking systems are usually used to trace bug findings and recommendations of the system.
- Maintenance – Once the product is released, the support staff frequently uses a bug-tracking database similar to the validation stage.

These phases all have unique forms with which to represent traceability. We should join them together. This can be accomplished through finding the similarities, not in the representation of the traceability, but instead through the data itself. So every level should have an owner, a description of the change or decision, and a time stamp as a minimum. The representations can now easily be represented inside a database. This must be done from the standpoint of the overall model so that each of the unique

representations can be preset with measures in which to link the data. This has each phase tied to each other through the traceability. This also must be driven from a project point of view, as if it comes from any one particular team. Otherwise, it may not carry enough emphasis to be taken seriously.

The best method to depict traceability through the different stages a software product goes through still needs further research. A possible mechanism to assist in that research could be an empirical study whereby nearly identical traceable facets are presented in different forms, and feedback is given to determine which different traceability form has the greatest value.

TRACEABILITY DEPTH/FOCUS

When it comes to the world of software and all things conceptual, there is always a challenge in the depth or level of explanation needed to convey optimal comprehension. This is especially true in the world of traceability. To achieve the correct focus level a top-down approach should be taken until the measurements achieve optimal ROI.

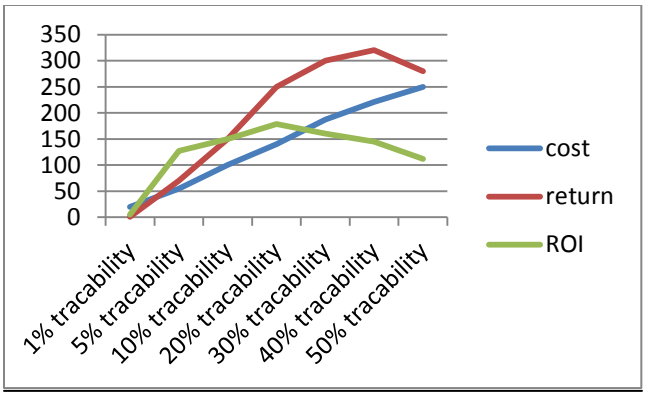

Figure 1: Traceability ROI

By top-down it is referring to taking the outermost view of the system and the highest point of reference, then continue to give more detailed explanation until the top level of ROI is achieved.

Though not proven via theorem, research has found that the benefit for added traceability usually follows a bell curve, as can be seen in **Figure 1 Traceability ROI**. Ideally, we want to add traceability at the peak of relative ROI. In Figure 1, the peak value is 20%. However, these values will differ in various projects.

THE TRACEABILITY MATRIX

The advocates of this new lifecycle model, other than those who speak of the amount of extra time and effort required, which is addressed later in this paper, are those who state that this is a glorified traceability matrix. Frequently associated with requirements is a well known entity called the "traceability matrix." This matrix usually ties customer requirements with low-level detailed requirements or functional tests. This matrix can act as well as a tool to tie final documents to system specifications giving a many-to-many relationship between the two. It is a very useful tool for requirements engineering, and it facilitates an ease in finding missing or unaccounted-for requirements specifications. The matrix is usually built with the requirements listed in the columns and the tests listed within the rows. Any non-empty box will declare a relationship between its corresponding requirements (column and the related document or test row). It is a requirement in some organizations such as the CDC, who actually publish their traceability matrix online [24]. There are several templates which can facilitate an enterprise in beginning to use the matrix; another example of these matrixes is illustrated in **Figure 2: Traceability Matrix**.

Requirements Traceability Matrix

Project Name: Software Registration System

Owner: Robert G. Nadon

Description: This project is a web-based software registration system to interface directly with the student allowing for self-registration

ID	Cust Spec	Funct Rqmt	Architectural reference	Documentation reference	Validation test	Status	Issue discovered reference	Comments
1	1.3.1	fcnl01	arch02, arch03	DDD pgs 2,8-15	7,18,21	58% run	Bug 1.075 mult entry access error, Bug 1.123 duplicate rec inconsist	Overall functional system achieving on-time scheduled target.
2	1.7.1	fcnl04	arch01, arch02, arch04	DDD ch 2	1-8,19,21	23% run	Bug 3.243 - Time delay during BU	
3								
4								

Figure 2: Traceability Matrix

It hopefully should be clear on how the traceability matrix, though it does give a form of mapping requirements to later stages, is different from the Traceable Lifecycle Model which will tie each phase or stage to the previous one with mapped traceability. The one thing the Traceable Lifecycle Model and the Traceability Matrix do have in common is that they are creating a mapping across different stages of the lifecycle.

Chapter 5: The Traceable Model

Now that we have gone into length on traceability and lifecycle model, we will construe a vision of the combination of the two. This leads us to the traceable model. The traceable model combines the advantages of the previously mentioned lifecycle models and intertwines traceability throughout.

Having the correct type of traceability has been proven to be vital. Though in virtually all cases some sort of traceability is extremely beneficial, this traceability is still frequently avoided or improperly done. Some of the reasons as to why are:

- The time and effort involved in adding the traceability
- Incorrect traceability either will be thrown away as useless or it will give incorrect insight, which makes things even worse as the traceability is misleading and actually destructive
- People generally do not want or like doing the traceability aspect of software, as it give traceability back to them as well as possible blame for a given action or artifact that they deliver
- This addition can, upon initial release, appear to have little or no added value, such as the old adage debug the code not the comments
- Different phases of a life cycle usually have different naming and representations for the traceability
- There are times when, for instance, a key customer is given a release with a key issue or bug and getting that issue resolved needs to happen as soon as humanly possible. This type of situation is where traceability is put on the back-burner and unfortunately once the problem is resolved is not again revisited.

- Economic, technical, and social factors all play a role in the reasons for the lack of traceability [25]

With this much going against traceability, why then is there the proposition that one should spend thorough time investigating exactly what traceability to add with the addition of a traceable origination factor for every artifact, in every phase of the system? This will be made clear through previously listed traceability benefits as well as the benefits of the new model, found later in the this paper.

Besides simply being the agreed upon right way to do things, there are many other advantages that can be obtained from this extra effort that was employed by doing all of this traceability. Software engineering's conceptual nature can only be made into a true engineering profession by quantifying the thoughts within the discipline, and a major part of that is accomplished by accurate and proper traceability. By having complete traceability throughout the project you have the upper hand in understanding each component and its derivation, evaluating each component for need and use, impact analysis for possible changes throughout the entire life of the system, further evolving the system by improving or maintaining it (fixing the bugs), and the ability to quantify the conceptual nature of software. But before going into the complete list of benefits of the new lifecycle, first is the definition of how it will look.

THE MODEL DEFINITION

To define the traceable lifecycle model, one should start by taking the current lifecycle model in use at your organization. Now all software is developed by following some sort of lifecycle model, be it documented or not. A recommendation is that if, there is not reasonably thoroughly detailed lifecycle model documentation for an organization that wants to move to a traceable lifecycle model, then this should be the first step.

Given the currently used lifecycle model documentation, one should abstract the common set of traceability attributes that can both be used to describe and name of the elements. For the example found in Appendix A, the previously listed seven traceable attributes were used as guides. It should be noted that all seven need not appear in every footprint. Using this procedure, you can see how these empty fields actually enlighten the software engineering into insights of the system and possible underlying problems.

So after ensuring that the current model is documented (call this step 0) we need to transform this to a traceable lifecycle model. So take your now documented software lifecycle, and go through the following steps. Also, the more details you have in your software lifecycle specification documentation, the better off you are going to be down the road.

Step 1: Start with a current well documented lifecycle model, and graphically map it out. For illustrative purposes, the spiral model will be the example lifecycle model, referenced in Section 3 and mapped in Illustration 2.

Step 2: Take this model and break it up into the various phases. Generally, all lifecycle models will be "broken up" or separated into the various phases as that is part of the definition of a lifecycle model, but we want to illustrate this more clearly. For the spiral example, it is not 100% clear what a phase is. It could be each quadrant of the spiral or each portion of each quadrant. Let's chose to use each portion of each quadrant. Though either would work, it should become clearer later on which choice would be better. The choice is primarily based upon the artifacts that are produced. So we now have a lifecycle that looks like the following:

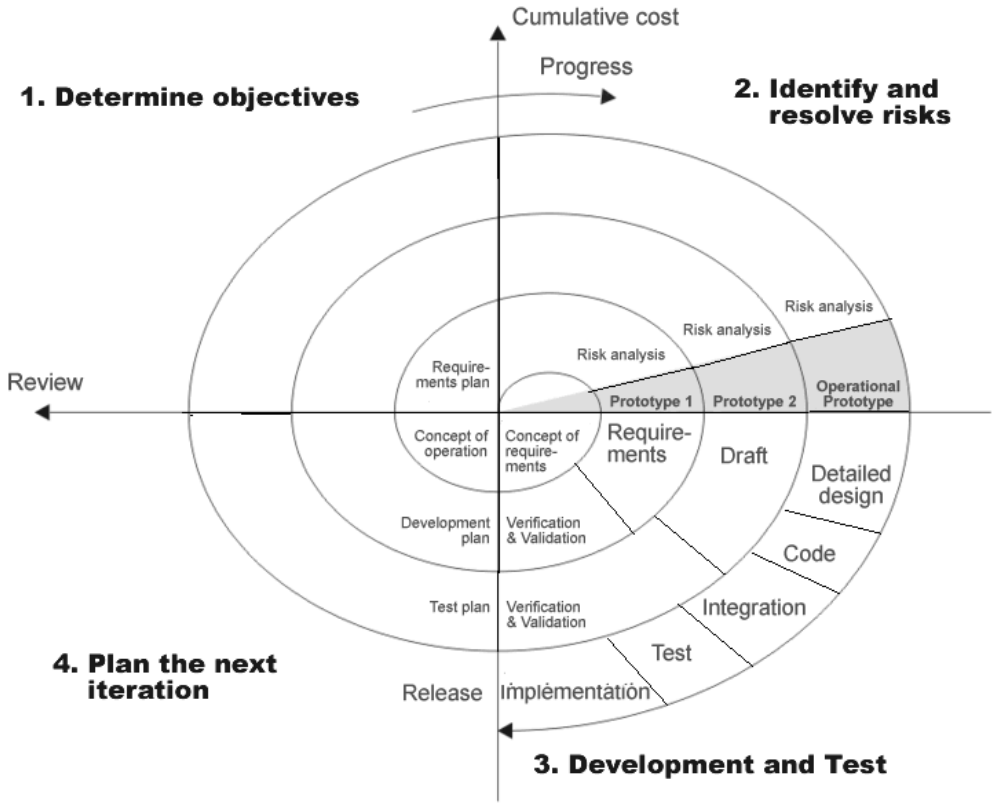

Illustration 4: Separation of the Phases

Step 3: From here we must ensure that each and every artifact that is supposed to be created during a phase is determined. The definition of the artifacts now comes into play. In high-level illustrations of lifecycle models, the various artifacts that are created during the phase are frequently not illustrated. However, these artifacts should be defined in any lifecycle model, as the artifacts that a phase produces are a significant part of the definition of that particular phase. So for illustrative purposes, details of the artifacts of each of the different phases will be avoided. Instead, simply use points or small stars for artifacts that are generated throughout the various phases of the spiral model. Taking this shortcut is understood as the various artifacts within a real life project are going to greatly vary, depending on the project and will be specified explicitly within

the traceability fields. With the addition of artifacts for each phase we now have a picture as follows:

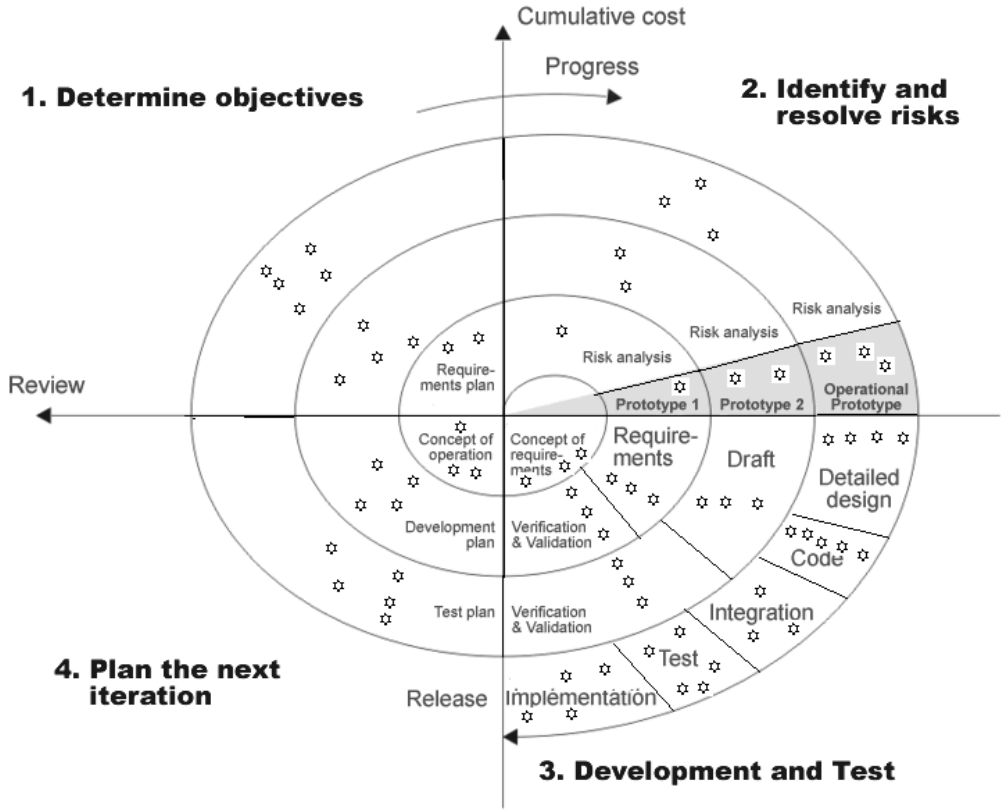

Illustration 5: Addition of the Artifacts

Step 4: This leads us to the final step of using the where field to connect the artifacts. To connect them we put into use the extra effort of the origination field. This was entitled as "***The Where***" for the example in Appendix A, and in certain cases we had to map other descriptive elements into ***the Where*** field. This mapping is a many-to-many mapping whereby an artifact can be originated from 0 or more previous artifacts and can be responsible for originating 0 or more future artifacts. After mapping the ***Where*** field by drawing a line from the artifact back to where it was originated, the final illustrated traceable spiral model looks as follows:

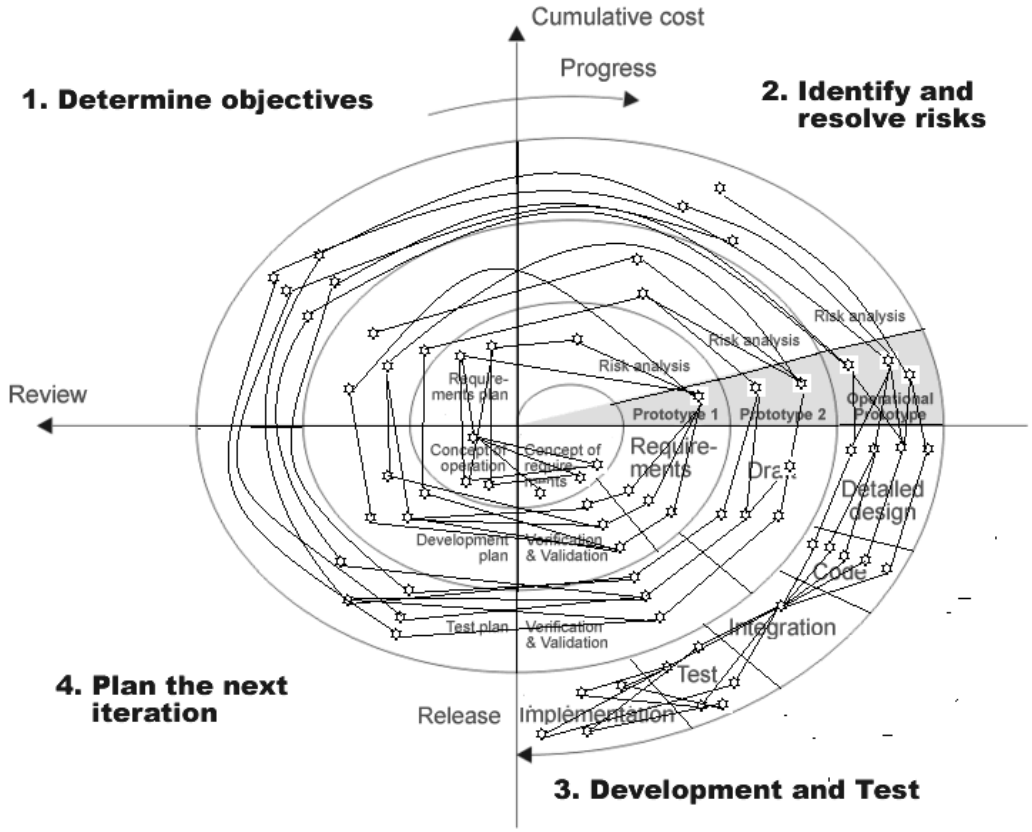

Illustration 6: Addition of the Links

Now without real artifacts and attachments the illustration is left without much insight. When implementing an artifact using the above traceability model, real-life models will have complete traceability defined, and a great deal of insight will be obtainable. This should become apparent with the Student Registration System example found later describe the paper.

When there is a transition from one phase to another, the later phase is usually defined, or at least guided by, the exiting artifacts from the previous stage. However, there are instances whereby a new artifact is generated, which had no bearing on the previous phase. This was illustrated as a new risk which ended up needing to create a new portion of the prototype and propagated itself from there. Furthermore, there are

artifacts which may jump a phase which is completely acceptable, as well as the artifact that simply does not have anything that is after it. This is the rarest case, as it begs the question of why was this artifact was created in the first place, if it is never going to be used again, but this does happen. Take for instance a portion of the system that was originally a portion of the system, but was later determined to be an unnecessary and/or unwanted part, so it was discarded then and there. Furthermore, it is frequently the case that there are artifacts that bridge things inside a phase, and this type of traceability should be captured as well and is illustrated by having artifacts point to each other within a phase above in the "Draft" sections of the spiral model.

The simple fact is that there are no hard fast rules about the traceability and the artifacts; just that as one stage leads to the next, there should be a traceability of each item that was evolved from a previous stage. Even though there are no rules, there are warning signs that can be read from the final picture. If there are any of the above listed inconsistencies, such as no follow-on artifact, this may in fact be a missing component that needs attention placed upon it. Furthermore, this can be used to depict basic dependency graphs and similar endeavors. If everything in a phase goes into one artifact, or if all artifacts coming out of a phase go to a single artifact, then those are points in the software lifecycle that have a high potential of being bottlenecks and areas of concern as well, and should be given the proper attention.

Why did we go through all of this extra work? The answer is listed some advantages of our new lifecycle model.

ADVANTAGES

According to one of the founders in software maintenance, Dr. Meir M. Lehman, "As software is revisited it becomes increasingly more complicated with declining

quality." [26] One of the primary reasons for this is that when a software project is revisited it must be re-conceptualized. The human mind is one of the most complicated pieces of machinery the planet has ever known. Given this, to find and rebuild the style and pattern that the previous engineer created, becomes an elusive and almost insurmountable task. This is the daily work of the tireless maintenance engineer. One accessory this engineer does have, especially if done correctly, is the traceability left by the original creator. This traceability can be invaluable; however, since software is conceptual by nature, it can also be very misleading if not done correctly or maintained throughout the entire lifespan of the project.

67% of a project's effort is spent on maintenance, which is more than all other phases COMBINED, [27] and this becomes an area that truly deserves attention. Since almost all projects need revisiting, either through desiring an updated version filled with improvements or through some manner of reference or reuse, we must take into consideration the maintenance and re-work effort that will undoubtedly occur. Traceability, though having many benefits, is especially beneficial for these efforts. For a project to have code that is never looked at again, anywhere by anybody, though it does exist, is almost an anomaly. The best way to truly illustrate the benefits of having a completely traceable model including origination is through a more realistic example.

The appendix lists a sample project created solely for this paper using a traceable waterfall model. Please see **Appendix A: Student Registration System** for further details. With no claim staked to be the world's greatest programmer, analyzer, nor architect/designer for that matter. Given that pat-on-the-back premise, any person, regardless of their technicality, should see some possible improvements they could make. With the added traceability more insight into the impacts these changes would have should also be apparent.

So start by mapping the traceability artifacts into their proper categories and graphically mapping them together.

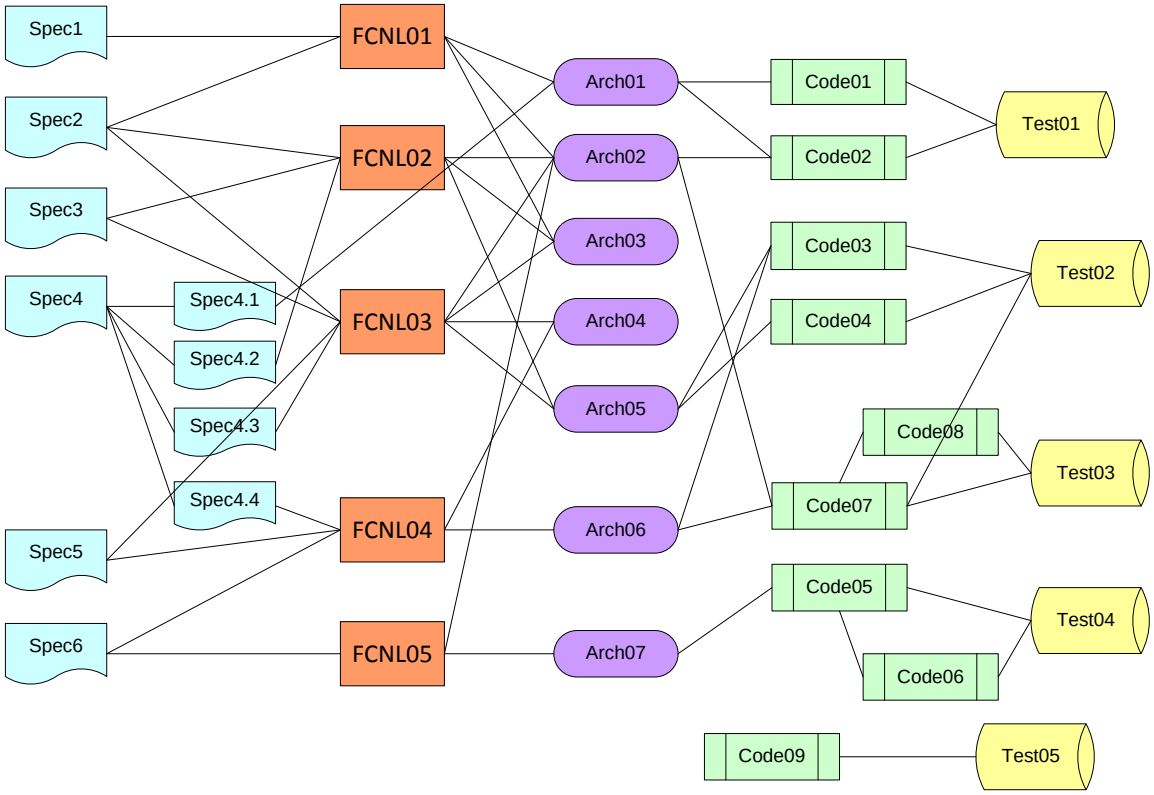

Illustration 7: Lifecycle Binding – Student Registration System

The traceability lifecycle model takes this binding and brings it throughout all stages of the software project. This traceability binding for the Student Registration System is illustrated in **Figure 10 - Lifecycle Binding – Student Registration System**.

Missing Links

Clearly visible in the illustration, when any component is not directly tied to the stage before it, a miss is found. For example, code segment 9 was not part of the requirements or architecture specifications. This module was the realization that courses

can be changed or removed and the system has no clear methodology to handle this. This is highly likely to cause problems in the overall system as the architecture and requirements don't specifically include this functionality. In real-life cases this may be due to missing pieces of software like this example or it could be control code that is needed to build the package or extraneous libraries that were necessary, yet not called out in any specification, which are all perfectly acceptable. Regardless of the underlying reason for a missing link, it signals the need to give the proper attention to ensure a problem is not encountered.

Also, if no exiting link is found to where this artifact did not affect anything in future stages, this may also be a problem. Looking at those things that have no link following them, we find that architecture modules 3 and 4 fall into this area. In our case, this is due to the fact that those items are pre-existing and so coding and testing them is not necessary which is perfectly acceptable. However, it may be the case that components that are not included in any future part of the system may illicit a missing functionality or testing.

Regardless of which direction or side the link is missing, it brings out an area that deems attention. Finding those areas that the misses are stemming from will provide a great deal of insight into where the overall system may need to be improved.

Phase Jumps

It is very possible to have links that jump across a phase or multiple phases. For instance, customer specification 4.1 does not have a requirements specification but it is found to be defined within architecture module 1. In the waterfall model, this almost always signifies a problem; however, in other models this may be expected behavior. For instance, within the Spiral Model there would be times that an artifact has no risks

associated with it yet it continues to survive throughout the lifecycle, in which case the link from this artifact would simply cross over the Risk analysis section or the spiral model. Regardless of the model, this needs to be taken into consideration and deems attention. Is there a reason for the jump or is it part of the lifecycle that one phase does not take the artifacts into consideration?

Fan-In and Fan-Out

Next, one should look at when there is a large fan-in or fan-out. Though not an indication of problems, they are key points in the project that engineers, especially the project manager, should be aware of. These areas have high potential for being bottlenecks and possibly delaying final release dates of the project. On our mini-demo, project the Student Registration System's ARCH02 module has a large fan-in and FUNC03 has a bit of a large fan-out. For ARCH02 this is due to almost every function of the system requiring authentication, while FUNC03 is adding a course, which is the fundamental part of the system. In more real-life situations, there can be even further insight in that if every architectural module goes into a certain section of code changes to that piece of code could have enormous impacts to the overall system. Regardless of in or out, conceptual knowledge can be obtained from these points of the project, and can have equally sized change impacts, regardless of the original lifecycle model.

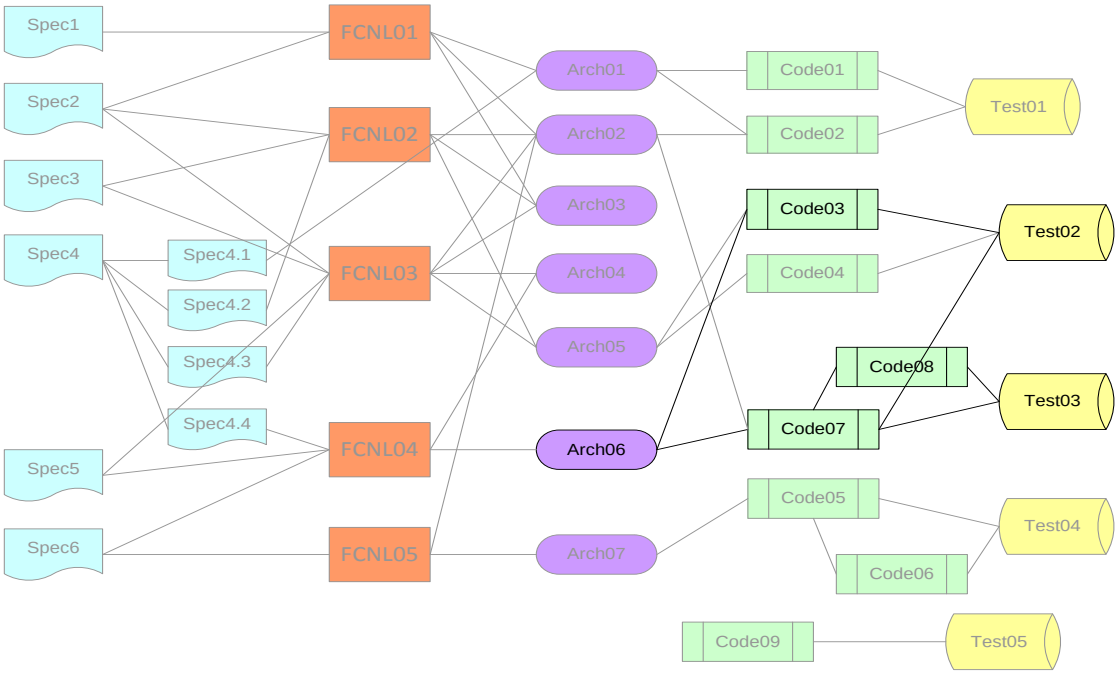

Illustration 8: Lifecycle Binding – Impact Analysis

Impact Analysis

Another great advantage of this new traceability is impact analysis. For instance, if we were designing a student course registration interface and upon initial prototype release it was determined that the students could only register for one class. Furthermore, this problem was root caused into the scheduler architecture module declaration of student-class as the data table primary ID. If the base layout matched that as seen in Illustration 7 and we have root-caused a problem with ARCH06. Then the impact of altering that design module can be seen in **Illustration 8: Lifecycle Binding –Impact Analysis**. From this it can be automatable and clearly illustrated how a change in this module could impact code segments 3, 7, and 8 test suites 2 and 3. We must be careful here and not forget about the previous stages as well. Following the link the other

direction we can see that possibly requirements 4 and specifications 4, 4.4, 5, and 6 lead up to this module and so they need to be revisited as well.

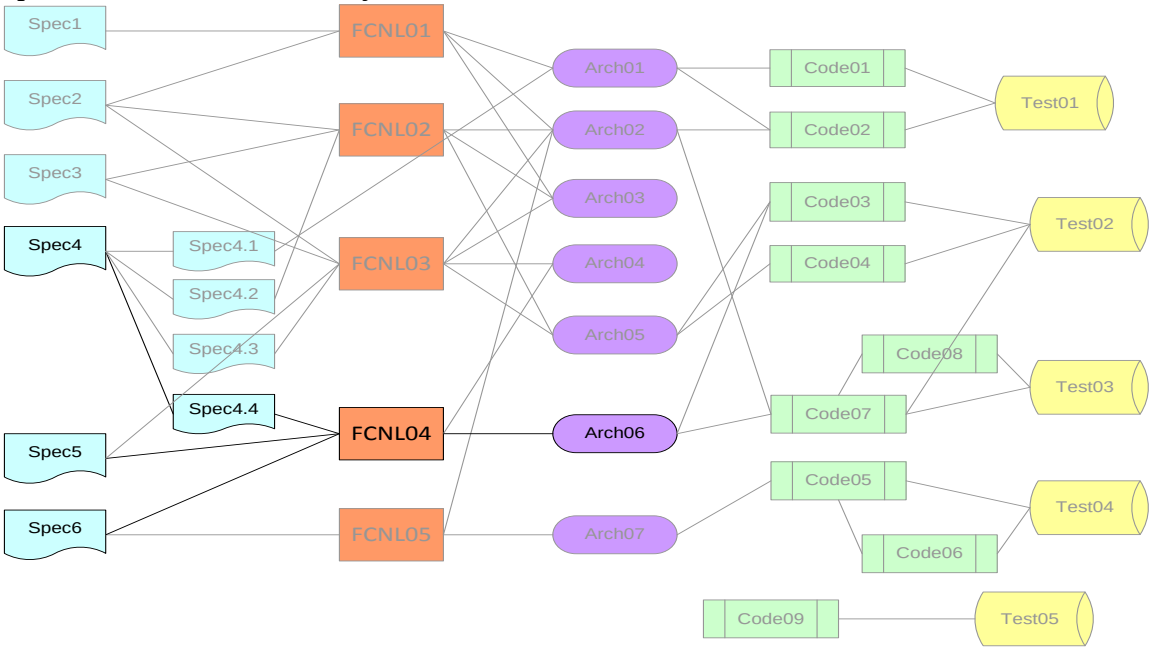

Illustration 9: Lifecycle Binding – Prior Stage Analysis

There are many reasons for going back to previous phases and keeping them consistent. If we only change the code, and not the architecture and design, then all of their benefits drift away until they are completely lost. This is known as *architectural drift* and is a very common problem with today's software engineering field. Furthermore, the lessons are not learned. For example, if the architecture has a miss and the architecture is never again revisited, one would presume that this type of miss would again happen in future architectures. Also, general comprehensibility and many of the advantages of the traceable lifecycle are also lost. If we keep the product whole and do due-diligence, in keeping all stages up to date with accurate and proper traceability, the benefit of using the Traceability Lifecycle at this point becomes astounding. For example, to determine who should be notified if the above change request was required we can change the name

parameter from ID to the owner. Doing this for the previously mentioned change, we can easily see that we should notify the student rep, university rep, and administrative rep of the specification team, and Robbie Requispec in requirements for post stage impact analysis. Also, Architect Robert G. Nadon, coders Kathy and Simon, and Freddy Functional tester must be notified as the change may require modification or at a minimum re-verification of their components.

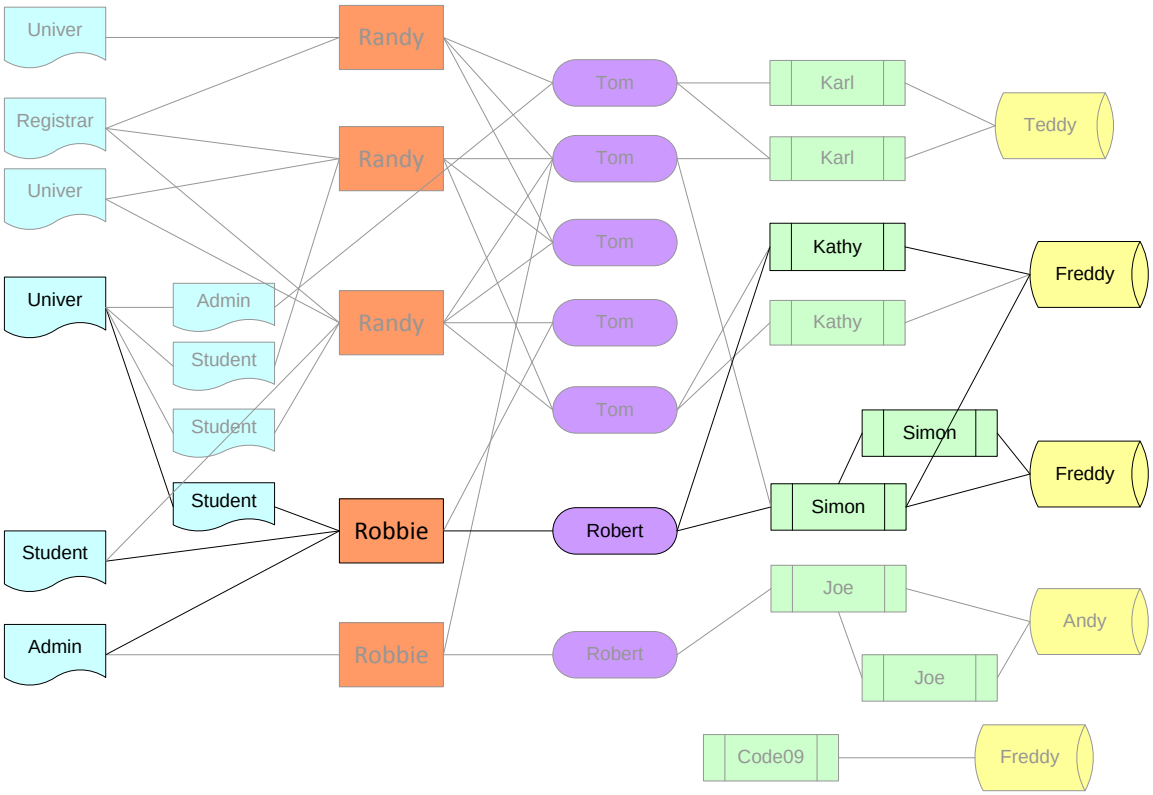

Illustration 10: Lifecycle Binding – Ownership Analysis

For further system understandability, changing the naming parameter from owner to component name, we get a clearer understanding of the system itself, as illustrated in Illustration 11.

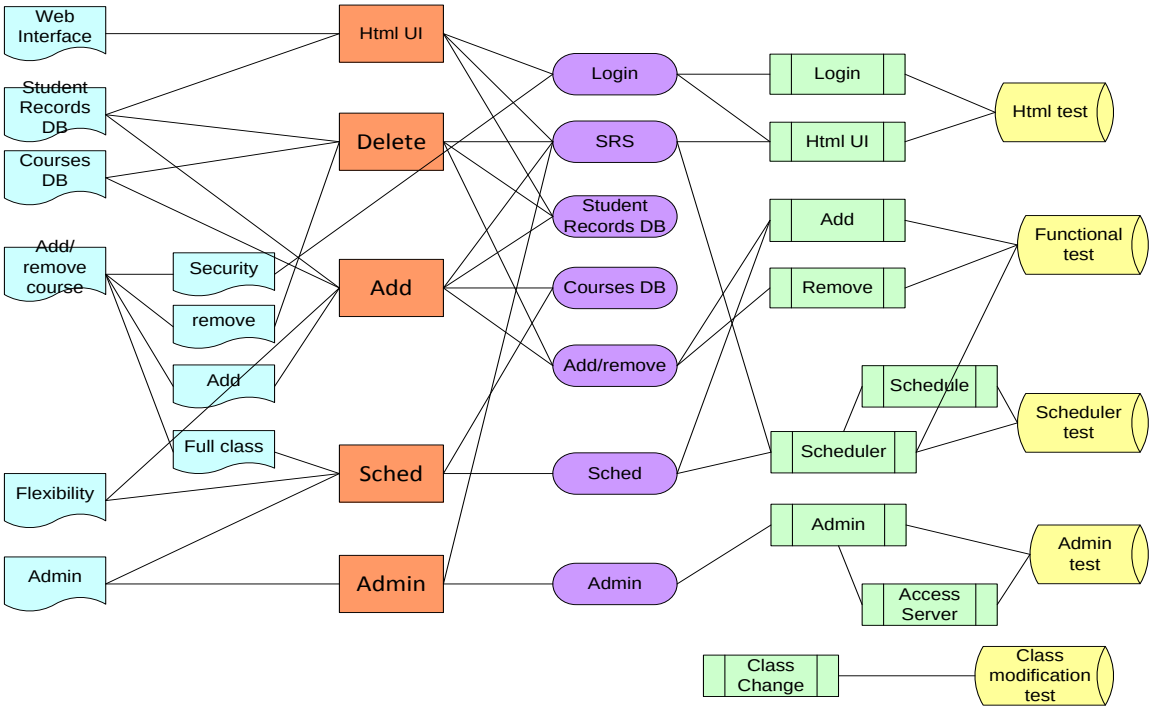

Illustration 11: Lifecycle Binding – Name Field

This model allows for many advantages. These advantages include things, such as:

1. It will allow for all phases to stay consistent and involved through the lifecycle, avoiding lifecycle drift and erosion. This greatly enhances the consistency of the product in that no phase can be "thrown away" as it does not go away.
2. It has evolutionary benefits in that each phase throughout the entire lifecycle is then traceable and accountable. When it comes time to alter anything, the traceable information will be of great use in illustrating overall the area that needs to be changed and its impact throughout the rest of the system.
3. It has the ability to illuminate the butterfly effect, or unseen minor changes that cause unseen detrimental results.
4. It clearly defines each stage, allowing for an easier method for capturing metrics.
5. It allows for resolution of bugs to be more easily traced to a root cause.

6. Traceability brings to light the troublesome areas of a software system, such as components that are written that do not tie back to a previous stage.
7. Documentation can be written in a more usable manner as each decision made has visible rationale.
8. Marketing is better prepared to explain why the product does what it does and this further helps promote the product's advantages.

There are still concerns understood with this model. First of all, it may be the case that after the preceding team completes the work on the given project they will need to move on to the next project. To help assist with this problem the amount of work in the phases following the original will be of smaller size as there will only need to be a verification that their work was not contradicted. With this new lifecycle model fully researched and defined, the extra effort is worthwhile, and should be actualized with ROI information.

STAGE VERIFICATION

This added value can be guaranteed by ensuring that consistency is prevalent throughout. To do this, none of the phases should have ending points until the project is complete and the final ending phase, such as the maintenance phase, comes to completion. This ending generally should only happen when another major release is put on the market. Above that, to transfer from one phase to the next all artifacts must have the verification through all traceable fields and variables completed. Furthermore, there must be a pass back for an "OK" given by the previous phase.

Ideally with this new traceable lifecycle power we want to ensure that this traceability is correct. The ability to "see into the creator's mind" was referenced earlier; however there is more to it than that. With traceability especially the interconnection of

the various stages usually involves a link between multiple creators within different groups. So, for example, a designer decides that the functional module that he or she has created was designed to satisfy requirements 2, 3, and 4. It would be nice to have the creator of those requirements – the one who documented them – also concur that the software does in fact satisfy the said requirements. Given these two pieces of information, first of all we must document the traceability so that we know what the creator believes to be the reason for origination of the new creation. Add to that, the creator of the artifact that pointed to this artifact should agree with the original creator's stipulation. This would lead to having a sign-off gate before every stage that requires the previous stage to concur with the new traceability. Ideally, all stages that link the entire documentation footprint to that point should concur that we have a working adequate representation of how the system has evolved to this point. To see this illustrated with the waterfall model of the proposed traceable sign-off model please see **Figure 3: Sign-off Procedure**.

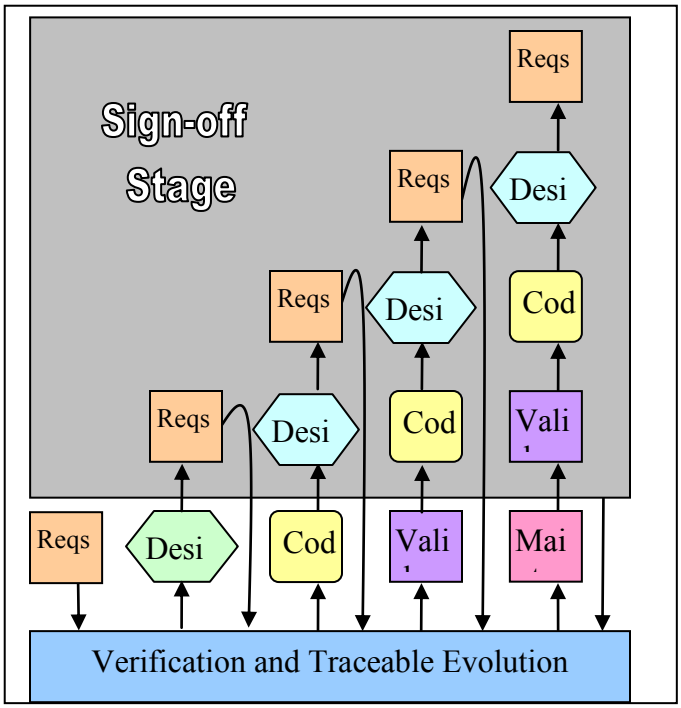

Figure 3: Sign-off Procedure

Though this sign-off stage may appear to be a great deal of added overhead, a quick sign-off to get to the next stage from all previous stages – no matter what type of model – is an incredibly useful and viable way to keep things on target and avoid the numerous problems with the famous throw-over-the-wall mentality that some organizations have taken. This mentality, is found whereby once the stage is complete, it is somewhat blindly handed to the next team. From there the current team then moves on to the next challenge, never again revisiting it. What frequently happens with this scenario is that the architecture team, for instance, takes time and creates this wonderful architecture, as far as architectures go, that would be admired by anyone else in the architecture profession. However, the designers do not take it as such a work of art. They take it, thinking those architects don't know what they're doing, and create a wonderful design with similar attributable qualities, but not in adherence to the referring architectural specifications. They then hand it to the coders who say, "Those designers are crazy," and continue to code what they believe to be the best solution. This has the effect of not only discrediting the accomplishments and goals of the previous phase but also dissolving a lot of the advantages of having a lifecycle model and pre-design and architecture steps to begin with. A sign-off stage would help to place a check to help alleviate this sort of mentality.

So with traceability now intertwined within our lifecycle model and a confirmation from the previous stage that things are still in alignment, we have a lifecycle that truly benefits from an abundance of conceptual insight. If this becomes standard it would be a start in bringing the field of software into a more engineering based discipline, and therefore be able to deliver consistent and reliable end products.

AUTOMATION WITH TM-CONNECT

Through use of an automated tool we can save time and get a high-level view of the complete system. A tool, ideally called TM-Connect and pronounced as "Team Connect," can be developed to grab the traceability information as long as it was done in a consistent manner. The tool could then illustrate the impact of system evolution by graphically showing the inter-connected pieces. This tool unfortunately cannot be a general tool that can be used for anyone attempting to employ the Traceable Lifecycle Model. It instead will be as unique as the traceability itself, and should be created if deemed to be a positive ROI. There are two basic portions that could possibly be automated. The first is collecting the traceable attributes and building them into a data structure. The second is taking this data structure and mapping it out graphically with the ***Where*** links used as lines connecting one picture to the previous one.

Please see **Illustration 12: TM-Connect SRS**, for an illustration whereby the SRS system was entered into a web-based tool. TM-Connect for this example will show both the previous-stages impact and the post-stages impact upon a mouse over of the given artifact. Given the traceable details about the different artifacts, its purpose is to know how to pull out the information place it into a data structure that uses the ***Where*** field to tie the pieces together. Essentially each artifact will be a figure of the picture and the origin or ***Where*** field will be a pointer or a line from the given artifact to the originator.

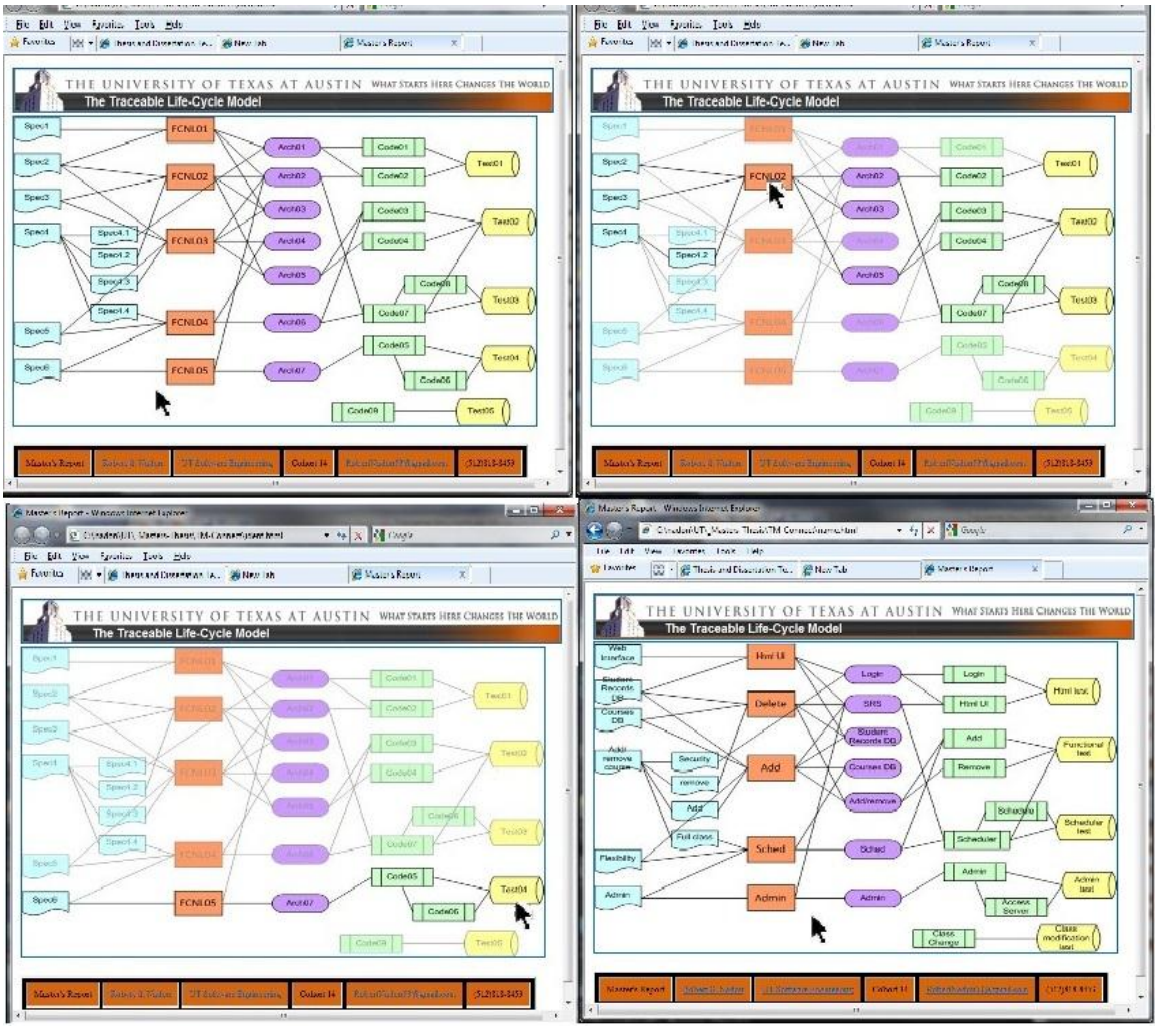

Illustration 12: TM-Connect SRS

There is a set of requirements that are necessary in order for the tool to work properly. They tie into the ability for TM-Connect to be able to understand where the traceability is within the artifact and how to pull that traceability out and place it in the data structures with the proper taxonomy. This can be accomplished either by manually selecting the pieces of traceability then placing it into the tool, or by creating a type of search engine with the ability to abstract the traceability fields. Having a glorified automated search program to seek out and abstract this information is an ultimate goal and would be a huge benefit to the software industry; however until this information is

stored in a standardized manner these search mechanisms are going to vary across different products and organizations.

Another requirement is that ***The Where*** field must only point to another artifact within the system (or be an empty field). For the appendix example, the ***ID*** field of a different cell within the system was used. This is part of the definition found within this field, and is necessary in order to establish a link from one artifact to another. There are artifacts that have no instantiating entity and this is perfectly acceptable. There are also those artifacts which do not instantiate anything beyond themselves, which too is acceptable, as previously discussed. The first stage of the lifecycle will frequently fall into the no-instantiator category. This is OK, and not all cells must even include a ***Where*** field. ***The Where*** field simply is dictating which previous artifact is responsible for originating the current artifact. There are also those that have no instantiatee which also was discussed in detail earlier in the paper.

Given the format of the traceability and searchable mechanism TM-Connect could be given set of flat files that contain the traceability with certain predefined parameters given, such as field notation, naming, and separation. It could then parse these files, and then using known text abstraction techniques, place the information into a data structure. An example syntax can be seen in Table 2 Traceable Variable Definition.

Variable	Specification
id	char[5]
who	char[25]
when	date
what	char[255]
why	char[255]
with	char[255]
where	*id[10] //array of ID pointers

Table 2: Traceable Variable Definition.

TM-connect can now be formed into a relational database table with the ***Where*** field being a link to related records and then display the interconnections, and this data structure can be read into a graphical tool that creates the visual graphs.

THE TRANSFORMATION

Transformation to a different lifecycle model is a major undertaking, too large for most enterprises to accept. Frequently the organizational structure is modeled after the lifecycle model that the software follows. If an organization follows some form of the Waterfall Model then the organization may be encapsulated as such where the design team and programming team may not only be in different departments, but may even be in different countries. So not only would we have to move to a new lifecycle model, but may in fact have to do a complete organizational change. Most who have been employed by a large corporation are all too familiar with the problems associated with the infamous

"re-org." However, a huge advantage of the traceable lifecycle model is that the base lifecycle is still followed.

The transformation to a traceable lifecycle model could also occur for a unique model that is proprietary, instead of the standard models defined in section 3. Since the basic definition of a lifecycle is that things happen in stages, the only requirement is that you have artifacts and the only impact is that you must take the existing model and add the correct traceability to it.

A strong recommendation is that a specialized traceability expert be hired for the initial change to the new model. This has the added advantage that a single person is overseeing all the traceability, and so this makes it easier to keep the overall system consistent. Without this consistency, there can be drift as the various stages define their own unique version of the footprint. From one stage to the next the change would hardly be noticeable, but if one were to go and compare the original footprint to the final maintenance-stage footprint then one might be surprised by the impact of the drift.

What must happen is to first define the data field necessary that will define the footprint of how your system will need to be traced. From there this standard set of traceability data could be transcribed as part of each stage or phase of the currently used model. Extra caution could then be made so that a transition from one phase or stage to the next could not occur until the traceable data was completed and stored. This could be easily incorporated into the given stage release documentation. With this, the current model that your organization uses can be transformed to the new traceable model. A further recommendation is that this transformation happen slowly and appear more as an evolution than a transformation.

Chapter 6: Summary

There has been industry emphasis placed upon requirements traceability and the importance therein; however, the paper has taken this to the next logical step: complete traceability for every phase and every artifact within that phase. To ensure this traceability is done, it has intertwined the traceability with the lifecycle itself. With this powerful new lifecycle some of the benefits include:

- ✓ Ability for problems to be resolved more quickly
- ✓ Ability to identify the impact of changes to avoid the butterfly effect
- ✓ Automated ability to view notification chains
- ✓ Enablement of the system to more smoothly evolve
- ✓ Giving each decision credibility and comprehension as rationale
- ✓ Overall system understandability

What is the value of this new comprehensibility to a future set of eyes altering the system? To be able to have the understanding of where each attribute within the system came from, and how it was derived would be nearly priceless. The project manager reaps giant benefits as well, as they now have the impact analysis to be able to instantly have a clearly defined workload of the list of various attributes across the lifecycle stages that will be impacted by giving change to any phase.

Finally, this paper proposed how this new model is achieved through evolution, instead of going through a complete model changeover. To truly change to this form of model, a mindset needs to change to appreciate the importance and usefulness of traceability. Further case studies and ROI measurements should continue to be taken on the power of traceability, not just until a product releases, but also long after following the product through its stages of evolution.

Appendix A: Student Registration System

Student Registration system

This example is a simplified student class registration system. It further assumes that there is a student system or database assumingly from the registrar, and a courses database. This system will simply provide an interface for students to be matched and assigned within the courses database. There is no financial part of this system, nor is there any ability to be placed on a waitlist. If the class is open the student can register, if it is closed (s)he cannot. Please note, I am not proposing that this is the ideal student class registration system, as a matter of fact, it is deliberately inadequate to help illustrate the power of good traceability.

Traceability field specification

ID – Unique field with a 4-char level description and 2-digit sequential number
Who – Define ownership and contact information
When – Date/time stamp
What – Description of implementation
Why – Rationale
With – Same-level components that interact
Where – Derivation description ID

Specifications from the customer:

Spec1. Must have Web-Based User and Administrator Interface

Who: University specification

When: 7/17/2010 10.31

What: Interface usage specification. The user interface must support standard web-based access with as few limitations as possible. This interface must support at a minimum Internet Explorer and Safari web browsers and be able to be run on Java 1.5 or higher.

Why: As the primary method of interface at most places has become the web, it is imperative that User and Administrator Interface be defined to work with other university based tools.

Spec2. Must interface with the pre-existing faculty/student database

Who: Registrar specification

When: 7/17/2010 10.37

What: An interface must be defined and made secure so that any information, including schedule, can be obtained from the existing database faculty/student database system. This should be used for ID information as well as user information.

Why: Current databases for storing information about faculty and students is currently functional and secure, so inventing the wheel is not necessary here. Maintaining a separate database could lead to inconsistency in records management.

Spec3. Must interface with the pre-existing courses database

Who: University specification

When: 7/21/2010 17.31

What: An interface must be defined and functional to interact with the current courses database. Instructors will add/delete/edit the courses from the courses database and the student registration system needs to use this course database to match students to courses.

Why: The course database system is preexisting and should interface with the new registration system. All information about the courses except the registration list will be stored in this system.

Spec4. Must be able to add and drop courses

Who: University stakeholder specification

When: 7/23/2010 11.18
What: The primary function of the system will be to add and remove courses from a given student's schedule

4.1. Must have security mechanism to ensure students can only access their own accounts

Who: Administrator stakeholder specification
When: 7/23/2010 11.20
What: Current 128-bit encryption along with mechanisms at both ends (f/s db and CRS) should be conducted to ensure authenticated access.
Why: Security is a large portion of the solution as confidential information, such as contact information and Social Security numbers are stored here.

4.2. Secure remove class mechanism

Who: Student representative specification
When: 7/23/2010 11.21
What: Must have a secure delete mechanism. This mechanism must allow classes to be removed at any time as there are no specifications on when a class can be removed. The system must ensure that when requested to remove a class from a schedule the request did not come by accident
Why: As classes fill up, this system must ensure that someone does not hack the system to remove another user to drop a class that is not his/hers. It was decided that no specifications be given to not allow someone to drop a course as this is, at this time, a no-charge university and so a class may be dropped any time including after the class has already begun.

4.3. Must have add-class ability which ensures that the same person is not required to be at two places at the same time.

Who: Student representative specification
When: 7/23/2010 11.25
What: Must have a mechanism to not allow courses to be added that conflict with time already reserved for a given student and semester. When scheduling class there must be a location factor and based on course location, back-to-back classes may also be blocked. If the course is not located at the same primary address then there must be a minimum 30 minute lag time between classes. Furthermore, this stipulates that the courses must have a maximum location distance of less than 30 minutes.

Why: The locations of courses can be varied and time must be given to ensure that a student has adequate time to get from one class to another.

4.4. Must have the intelligence to inform students of full classes

Who: Student representative specification

When: 7/23/2010 11.18

What: A global tally must be maintained for each class so that only the correct number of students may register for a class. There must be measures to ensure that if two students request a class, with but one seat left, at the same time that both do not get the class.

Why: There is no current waitlist and there is no overbooking such as an airline. Classes are filled until capacity is reached at which time there should not be allowed additions. This will allow adequate course material to be available and class location to be adequate.

Spec5. Must have a flexible ability to assist students in maintaining manageable course schedule

Who: Student representative specification
When: 7/23/2010 11.18
What: An addition mechanism must be put in place that has certain rules that do not allow a single student to schedule too many classes. There should be a default that allows students to take at most either 4 classes or 12 class-hours a semester. This default may be overridden with special approval.
Why: There are those who are overzealous and may schedule more than they can handle. There are also those who can handle the extra load, so a special meeting with appropriate university personal must be conducted prior to allowance of more than 4 classes or 12 class-hours a semester.

Spec6. Administrative module must be able to be run remotely and should not need more than one part-time dedicated engineer to maintain (Administrator stakeholder specification)

Who: Administrator stakeholder specification
When: 8/11/2010 16.12
What: Administrative module must have a secure interface that allows administrators and only administrators to be able to do basic user and system administration for a remote location. A standardized secure shell interface

should be provided. The administration needs the ability to see what is happening with the systems as well as the ability to stop and restart the system.
Why: It was determined that the administrators will be located in an off-site location from the systems and so for convenience the administrative module should be able to be securely used remotely.

Requirements – Functional
Function ID: FCNL01

Function Name: Student information viewing

Description: Allow student or authorized personnel viewing of student's schedule and personal information.

Rationale: The system is designed to have students populate their schedules for a given period of time and so viewing what has already been accomplished is a requirement.

Inputs: Student ID

Outputs: Student personal information, student completed course list for the given semester

Interface: Student records database

Pre-Conditions and Triggers: Student enters system, requests catalogue update

Post Conditions: Student gets personal information

Constraints: Student ID verification

Stakeholders: Student

Session ID: Functions_SRS_082510_9:28, Randy Requirement

Origination: Spec 1, Spec2

Function ID: FCNL02

Function Name: Student Course Delete

Description: Allows a student to delete a course from the class list

Rationale: There are two primary functions of the system to add and delete a class. This is the later.

Inputs: Student ID, Course ID

Outputs: student catalog of enrolled classes

Interface: Student registration system

Pre-Conditions and Triggers: Student requests class delete

Post Conditions: Course listing system updated without the new class for the given student

Constraints: Student ID verification, Not past inability to drop date

Stakeholders: Student

Session ID: Functions_SRS_082510_10:14, Randy Requirement

Origination: Spec 2, Spec3, Spec4.2

Function ID: FCNL03

Function Name: Student Course Add

Description: Allows a student to add a course to the class list

Rationale: There are two primary functions of the system to add and delete a class. This is the former.

Inputs: Student ID, Course ID

Outputs: Updated student catalogue

Interface: Student registration system, Scheduler, Course listing system

Pre-Conditions and Triggers: Student requests class add

Post Conditions: Course listing system updated with new class for the given student

Constraints: Student ID verification

Stakeholders: Student

Session ID: Functions_SRS_082510_9:42, Randy Requirement

Origination: Spec 2, 3, 4.3, 5

Function ID: FCNL04

Function Name: Scheduler

Description: Ensure that there are no conflicts within the student's schedule

Rationale: As per the specifications a conflict scheduler was determined to be a unique function within the system and not be tied to adding a class for flexibility purposes.

Inputs: Student schedule, Course ID

Outputs: Boolean (True = no conflicts, False = conflict ➔for Class ID)

Interface: Student registration System, Course listing system

Pre-Conditions and Triggers: System requests to check a class for Student ID

Post Conditions: N/A

Constraints: Only the system can call the scheduler

Stakeholders: Student, Developer

Session ID: Functions_SRS_082510_10:34, Robbie Requispec

Origination: Spec 4.4, 5, 6

Function ID: FCNL05

Function Name: Administrator Module

Description: Allow administrators to ensure proper system function and be able to restart the system

Rationale: Administration was grouped into a single functional specification as all three operations are remote calls to the underlying hardware system

Inputs: Admin ID

Outputs: Notifications, process lists

Interface: Student Registration system, administration module, Student records database

Pre-Conditions and Triggers: Administrator enacts administration page

Post Conditions: System halt, system restart, system remain in same state

Constraints: Only the users having administrator privilege can access this area

Stakeholders: Administrator, developer

Session ID: Functions_SRS_082510_11:01, Robbie Requispec

Origination: Spec 6

Architecture

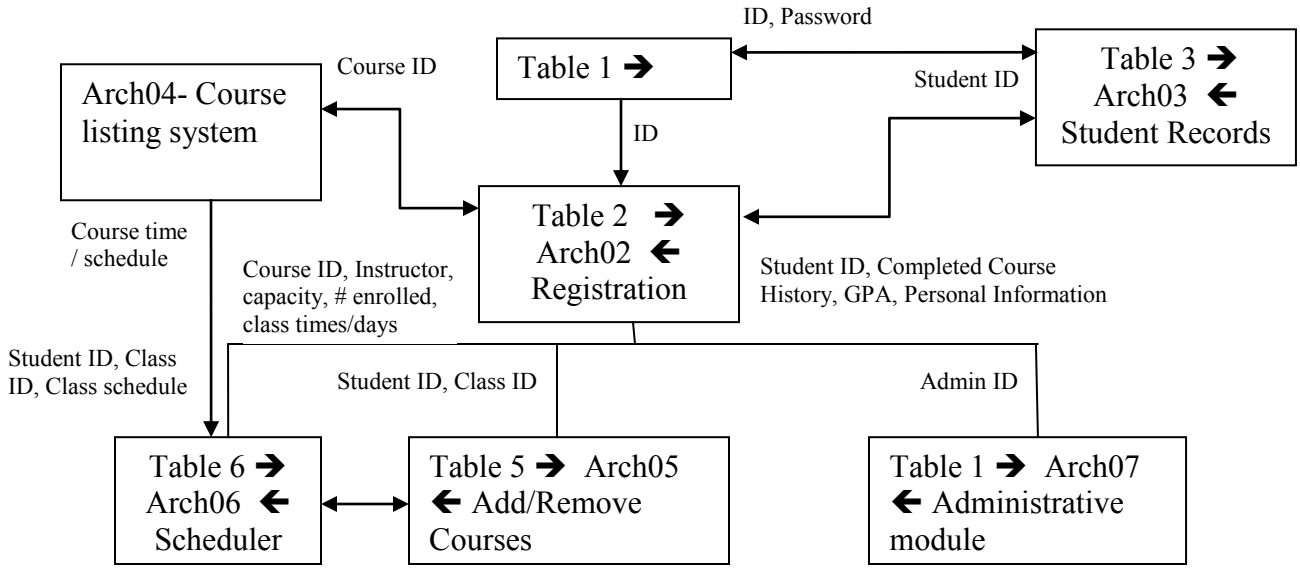

Table 1 ➔ Arch01 ➔	
Title	Login
Owner	Tom Architect 08/27/2010 06:55
Reference	Initial log-in interface
Interaction	Student Records Database, Student Registration System
Origin	FCNL01,SPEC41
Reason	Direct login into the system will be conducted though IDs will be stored in the student records database

Login system will:

1) Receive the login
2) Verify with Student Records database
3) Send verified IDs to SRS

Table 2 ➔ Arch02 ➔	
Title	Student Registration System
Owner	Tom Architect 08/26/2010 14:55
Reference	Main interface to students that will manage course registration and enrollment. This is a super-cell that will run the scheduler and the course add/delete functions
Interaction	SRS Add/Delete, SRS Scheduler, Admin Module, Course Listing System, Student Records Database, Login Module
Origin	FCNL01, FCNL02, FCNL03, FCNL05
Reason	The primary interface into the system will be sectioned into a unique module to

allow for the interface to be sectioned off

Student registration system will:

1) Receive the login
2) Display originating home pages
3) Pass though administrative operations
4) Coordinate course additions and removes

Table 3 → Arch03 ←	
Title	Student Records Database
Owner	Tom Architect 08/27/2010 07:51
Reference	Existing database containing a complete record of all users on the system and all relevant information about the user.
Interaction	Login, Student Registration System
Origin	FCNL01, FCNL02, FCNL03
Reason	This is a pre-existing system that the SRS interacts with and needs to be included in the architecture schema.

Student Records Database system stores complete student information including all courses taken in the past and currently registered for as well as any personal information on file. This is a SQL-based database that is accessible by this SRS system.

Table 4 → Arch04 ←	
Title	Course listing system
Owner	Tom Architect 08/26/2010 12:32
Reference	External system that holds all courses and their attributes. This is an external module added to the arch for interfacing purposes
Interaction	Student registration system, SRS Scheduler
Origin	FCNL03, FCNL04
Reason	This is a pre-existing system that the SRS interacts with and needs to be included in the architecture schema

Course listing system interface upon receiving a course ID; will return the complete set of attributes about the course.

Table 5 → Arch05 ←	
Title	Courses add/remove
Owner	Tom Architect 09/02/2010 10:35
Reference	Ability to add and remove class from a student's schedule.
Interaction	Scheduler, Student Registration System
Origin	FCNL02, FCNL03
Reason	The course add/remove is a main portion of the system and are keep as a single module as they are closely related and should be kept in the same module

The courses add/remove feature will add, if allowed by the scheduler, a course to the student's class listing, or delete a course.

Table 6 → Arch06 ←	
Title	Scheduler
Owner	Robert G. Nadon 09/02/2010 10:19
Reference	The scheduler will ensure that no conflicts occur in the student's schedule. Given a schedule and a course ID, the scheduler can have various rules tied to it about distance and time between classes.
Interaction	Course Listing system, Student Registration System
Origin	FCNL04
Reason	By dividing the scheduling information into a separate module this allows for modularity as this portion of the system is likely to change.

The scheduler will be a rule-based mechanism to allow for a class listing and course information to be passed and from the rules a Boolean value, true – "class fits into schedule" or false – "class does not fit into schedule," will be returned.

Table 1 → Arch07 ←	
Title	Administrative module
Owner	Robert G. Nadon 09/02/2010 10:45
Reference	The administrative module will have complete administrative authority over the system.
Interaction	Student Registration System
Origin	FCNL05
Reason	The Administrative module is the other portion of the system and all administrative tasks are kept in a separate module as access should be constrained.

The administrative modules give the super control over the entire system. This will allow the ability to make new rules within the scheduler, ensure login and ID authentication is accurate, and have the ability to bring up and down the entire system in a graceful manner.

Code/sudo-code : ➔Sudo-Code – Functional 🡐

\\\
\\ ID: Code01
\\\
\\ Name: Login
\\ Who: Karl Koder
\\ When: 10/23/2010 12.08.22
\\ Why: Secure interface and login mechanism is required as to ensure proper and confidential access
\\ With: Student records system
\\ Where: Arch01
\\ What: Simple HTML interface to request and verify through the SRD login-ID/password

Global New Sched Type Class-Schedule

Login (StudentID, password)

New FailMsg Type Message
If [Send Student_Records (StudentID, password) = True]
Then
Class_View (StudentID)
Else
Print <html FailMsg>

\\
\\ ID: Code02
\\
\\ Name: User Interface
\\ Who: Karl Koder
\\ When: 10/23/2010 14.55.27
\\ Why: Web-based system access was the first requirement and it was determined that there are two primary views – the student and the administrative view – so based upon login credentials it was determined to give two different, predefined views
\\ With: code01
\\ Where: arch01, arch02
\\ What: html front end

Print <html SRS header>;
If (ID=admin) then
Admin_View(ID);
Else
Class_View(ID);

Class_View (ID)

Sched = Send to Student_Records (View: StudentID);
Print <html Sched >;

Admin_View (ID)

```
        Print <html Admin>;
Print <html SRS footer>;
```

\\
\\ ID: Code03
\\
\\ Name: Add class
\\ Who: Kathy Koder
\\ When: 10/23/2010 14.18
\\ Why: It was decided that since the scheduler has the smarts to determine if a class is adequate for scheduling ii should also send the request to the student registration system to add the class.
\\ With: code02
\\ Where: arch05, arch06
\\ What: The class is retrieved from the courses database, passed to the scheduler for scheduling coordination who will either schedule the class or pass back why it was not scheduled, then the results of the operations are shown to the user.

Class_Add (StudentID, CourseID)

```
        New Class-conflict Type Message
        New Class_Info Type Class-Information-Array
        Class_Info = Send to Course_Listing_System (CourseID)
        Sched = Send to Student_Records (View: StudentID)
        If [Send to Scheduler (Sched, Class_Info)]
        Then
                Print <html 'Send to Student_Records (Add: StudentID, CourseID)'>
                Return(0)
        Else
                Print <html Class-conflict>
                Return(-4)
```

\\
\\ ID: Code04
\\
\\ Name: Remove class
\\ Who: Kathy Koder
\\ When: 10/23/2010 12.08
\\ Why: There are no requirements other than authentication for deleting a course. Since to get to this point all authentication has been verified, the delete operation is simply sent directly.
\\ With: student records
\\ Where: arch05
\\ What: See if class is in the student's Schedule and if so remove it.

Class_Delete (StudentID, CourseID)

```
   Sched = Send to Student_Records (View: StudentID);
   If (CourseID in Sched)
   then
         Sched = Send to Student_Records (Delete: StudentID, CourseID)
         Print <html Sched >
         Return(0)
    else
         FailMsg= "Class CourseID not found in StudentID schedule"
         Print <html FailMsg>
         Return(-3)
```

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\ ID: Code05
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\ Name: Admin
\\ Who: Joe Admin
\\ When: 10/26/2010 9.14
\\ Why:  It was decided that a separate function would be required to perform the
operations and the calling operations for security and for modularity as the low level
commands may change; however, the interface to those should stay the same.
\\ With: code06
\\ Where: arch07
\\ What: A set of three actions was determined to be needed for the administrative panel.
Those are a system halt, a system reboot, and a listing of the open connections the system
currently has.
```

Admin (AdminID, password)

```
         New FailMsg Type Message;
         If [Check (AdminID, Password) = True]
         Then
                  Print <html Admin Page>
         Else
                  Print <html FailMsg>
Adminview()
         if <select adminview = show connections>
         then
                  Connections=AccessServer.connections() ;
                  Connect-html= Parse(Connections);
                  Print <html $connect-html >
         else if <select adminview=reset server>
         then
                  AccessServer.halt(restart)
         else if <select-adminview=shutdown server>
```

```
        then
                AccessServer.halt(shutdown);
        else
                Print <html Admin Page>

\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\ ID: Code06
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\ Name: AccessServer
\\ Who: Joe Admin
\\ When: 10/26/2010 10.01
\\ Why: It was determined through surveys that a 10 minute message warning about the
impending restart/halt would be adequate and proper warning.
\\ With: code05
\\ Where: code05
\\ What: Commands to launch the various admin functions including systems connection
list, shutdown, and restart.
AccessServer(command)
Connections=Get.system(connections)
If (command == halt)
Then
        For each Connections
                Send-message ("system going down in 10 minutes")
        Pause(600)
        Powerdown-system
Else if (command == restart)
        For each Connections
                Send-message ("system going down in 10 minutes")
        Pause(600)
        restart-system
else if (command==connections)
        return (Connections)

\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\ ID: Code07
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\ Name: Scheduler
\\ Who:  Simon
\\ When: 10/27/2010 06.08
\\ Why: The only operation the scheduler needs to do is to add a class; there were two
primary reasons why not to schedule a class. 1. Class is full. 2. Class is conflict.  For the
rules of conflict, the class.location was used which, if in the same building, will be
identical and allow students to take back-to-back classes.
```

\\ With: code03, student records, code08
\\ Where: arch02,arch06
\\ What: Check for full classes, then if not full, check for scheduling conflicts. If all goes well passes the class and student to student records to schedule the class.

Scheduler (Schedule, Class)

```
	int max=Class.Getmax(Class);
	int load= Student_Records.enrolled();
	if ( load == max)
	then
		class-conflict= "Class Full";
		Return (-1);
	else
		if Schedule.available(Schedule, Class)
		then
			Class prior-class=Schedule.prior (Schedule, Class)
			if prior-class=none
			then
				Student_Records_AddClass (Class)
				Return (0)
			else
			For each Class.day
			  if (prior-class.end(day)=Class.beg(day))
			  then
				if (prior-class.location != Class.location)
				then
					class-conflict= "Prior class adjacent";
					return (-2);
			Next Class.day
			Student_Records_AddClass (Class)
			Return (0)
```

\\
\\ ID: Code08
\\
\\ Name: Schedule
\\ Who: Simon
\\ When: 10/23/2010 12.08.22
\\ Why: It was determined that a separate function be created to check if classes are back to back as this may be replaced by pre-existing calendar programs in the future so flexibility for this replacement is pre-baked.
\\ With: code07
\\ Where: code07

\\ What: There are two possibilities, either there is a class already scheduled at the time of the class or there is one adjacent. If neither of these two conflicts does occur, a simple no conflict is returned.

```
Schedule (Schedule, Class)
	available ()
		for each Class.day
			if (span(Class.beg,Class.end) = free )
			then
				return true
			else
				return false
	prior()
		while Class.time=free
			time=time-30 minutes;
			if (Student_Records.Schedule(time) != free)
			then
				prior-class=getclass(time);
				break
		return "none";
```

\\

\\ ID: Code09

\\\

\\ Name: Class Change

\\ Who: Cody Coder

\\ When: 10/25/2010 12:51:53.23

\\ Why: If a class changes, this is needed to allow for the altercation and notifications purposes. It was found during implementation that the possibility of having a class altered and removed was not accounted for.

\\ With: code04, student records database

\\ Where:

\\ What: Called when a class is either altered or removed, this function will get the list of students enrolled in the given class and send out a message. There are only two possible changes, the class is either altered (day and time/location/both) or canceled.

```
Class-Change (class)
	Student_Records (students, class)
	For each students (get-enrolled(class))
		Class-Delete (student,class)		If (reason==deleted)
		then
			Send message (student, "Class $class  is no longer availabe")
		Else if (reason==changed)
		Then
```

Send message (student, “Class $class has changed, you must re-enroll in this class”)

Test cases

ID: test01
Who: Teddy html-tester
When: 10-28-10
What: html-based test case to allow for consistent html testing and function and no dead links. This testing portion will be hands-on and not automated due to the tricky nature of web interface. Automatic test will also be run to simply ensure no dead links.
Why: the html testing portion was done as a separate step to isolate html-specific problems, such as dead links and poorly viewable graphics
Where: code01, code02
With:

ID: test02
Who: Freddy Functional tester
When: 10-28-10
What: functional based test allowing for addition and deletion of classes from a variety of sources with a variety of classes and students. Additionally, a stop watch will be put on all tests and the timing published. This portion shall have automated test sequences run with ability to add regression tests directly to it for continuation purposes.
Why: The important thing in this area is basic functionality along with speed aspects, so this section is the area where automated testing must be created and run prior to every release phase, no matter how small the change.
Where: code03, code04, code07
Who: Andy Admin
When: 10-29-10
What: All three administration duties are to be tested with various dummy loads on the system to ensure everything goes according to plan.
Why: this was a late realization that was added at code time and so must be thoroughly tested.
Where: code09
With:

ID: test03
Who: Freddy Functional Tester
When: 10-28-10
What: Specific special cases to ensure proper function of the scheduler sent directly to the scheduler.
Why: The complex rules set needs to be ensured to function properly, so it was isolated and thoroughly validated.
Where: code07, code08
With:

ID: test04

Who: Andy Admin
When: 10-29-10
What: All three administration duties are to be tested with various dummy loads on the system and ensure everything goes according to plan.
Why: Administrative area is a unique area and so should be sequestered and tested.
Where: code05, code06
With:

ID: test05
Who: Freddy functional tester
When: 10-30-10
What: A specific test was conducted to find out the ramifications of course modification and removal.
Why: This was a late realization that was added at code time and so must be thoroughly tested.
Where: code09
With:

References

[1] Brooks, Fredrik 1987, **No Silver Bullet: Essence and Accidents of Software Engineering** http://www.lips.utexas.edu/ee382c-15005/Readings/Readings1/05-Broo87.pdf

[2] Hilderman, Vance 2009, "**Traceability per DO-178B & DO-254**"

[3] Christensen, Mark J; Thayer, Richard H. 2001, **The Project Manager's Guide to Software Engineering's Best Practices**. IEEE Computer Society Press Order Number BP01199 ISBN 0-7695-1199-6

[4] Scacchi, Walt, 1987 **Models of Software Evolution: Lifecycle and Process**. SEI Curriculum Module SEI-CMI-10-1.0

[5] Niquette, Paul 1995. **"Softword: Provenance for the Word 'Software'".** http://www.niquette.com/books/softword/tocsoft.html. adapted from *Sophisticated: The Magazine* ISBN 1-58922-233-4

[6] DataMonitor, 2008. [DataMonitor - Abstract from Global Software Industry Guide - 2008]

[7] Lehman, Mier M 1996, **Laws of Software Evolution Revisited,** pos. pap., EWSPT96, Oct. 1996, LNCS 1149, Springer Verlag, 1997, pp. 108-124 [PDF] http://www.manageability.org/blog/stuff/the-8-laws-of-software-evolution

[8] Biffl, S.; Aurum A.; Boehm, B., Erdogmus, H.; Grunbacher, P. 2005 **Value-Based Software Engineering** Springer-Verlag Berlin and Heidelberg GmBH & co KG, Dordrecht, ISBN:9783540292630

[9] Knoernschild, Kirt, 2008, **Traceability & ALM – Fact or Fantasy**. http://apsblog.burtongroup.com/2008/10/full-lifecycle.html

[10] Certification Authorities Software Team (CAST), 2003. Position Paper CAST-18 Reverse Engineering in Certification Projects Completed June 2003 http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-18.pdf

[11] Baldwin, Diana, 2001. **Software Development Life Cycles: Outline for Developing a Traceability Matrix**, AccuReg Inc http://www.regulatory.com/forum/article/tracedoc.html

[12] ASI datamyte **Traceability and Lifecycle Management** http://www.asidatamyte.com/ftpgetfile.php?id=63

[13] Héctor García, Eugenio Santos, and Bruno Windels, 2008. **Traceability Management Architectures Supporting Total Traceability in the Context of Software Engineering**, http://www.foibg.com/ibs_isc/ibs-04/IBS-04-p02.pdf

[14] Kannenberg, Andrew **and** Saiedian**,** Dr. Hossein, 2009. **Why Software Requirements Traceability Remains a Challenge.** Crosstalk, the Journal of

Defense Software Engineering.
http://www.stsc.hill.af.mil/crosstalk/2009/07/0907KannenbergSaiedian.html

[15] Sink, Eric 2008. **Traceability Management Architectures Supporting Total Traceability in the Context of Software Engineering** "What is ALM? Traceability" http://www.ericsink.com/alm/traceability.html

[16] Egyed, A., 2006. "**Tailoring Software Traceability to Value-Based Needs**," in Value-Based Software Engineering (VBSE), editors: S. Biffl, A. Aurum, B. Boehm, H. Erdogmus, and P. Grünbacher, Springer Verlag, pp. 287-308.

[17] Gardner, Ed 2006. **Addressing the Grand Challenges for Traceability**. http://www.traceabilitycenter.org/cfp

[18] International Organization for Standardization. 2007 **Software Engineering – Metamodel for Development Methodologies** ISO/IEC 24744:2007. ISO 900.

[19] Boehm, Barry W., 1988. "**A Spiral Model of Software Development ... and What Designers Think**", Communications of the ACM, March, 28(3), 300-311 **http://upload.wikimedia.org/wikipedia/en/3/33/Spiral_model_%28Boehm%2C_1988%29.png**

[20] Courtesy of Software QA Testing
http://softwareqatestings.com/images/stories/common/V-Shaped_Model.JPG

[21] Krasner, Dr. Herb, 2010. **SE Lifecycle Models and Metrics.** SE Measurement, EE382V Measurements UT

[22] Hamann, Lars, 2010. **A UML-based Specification Environment**. Use Version 2.6.0: http://www.db.informatik.uni-bremen.de/projects/USE/#support

[23] Conklin, Jeff and Begeman, Michael L. **gIBIS: a hypertext tool for exploratory policy discussion.** CSCW '88 Proceedings ISBN:0-89791-282-9

[24] CDC Requirements Traceability Matrix
http://www2.cdc.gov/cdcup/library/templates/CDC_UP_Requirements_Traceability_Matrix_Template.xls

[25] Asuncion, Hazeline and Taylor, Richard N. 2007. **Establishing the Connection Between Software Traceability and Data Provenance**, ISR Technical Report # UCI-ISR-07-9

[26] Lehman M M, **Uncertainty in Computer Application and its Control Through the Engineering of Software**, J. of Software Maintenance: Research and Practice, v. 1, n. 1,Sept. 1989, pp. 3 – 27

[27] Schach, R. 1999, **Software Engineering**, Fourth Edition, McGraw-Hill, Boston, MA, pp. 11.) Add to that, over 80% of the maintenance effort is used for non-corrective actions (Pigosky 1997) Thomas M. Pigoski, Practical Software Maintenance: Best Practices for Managing Your Software Investment, John Wiley & Sons, Inc., New York, NY, 1996